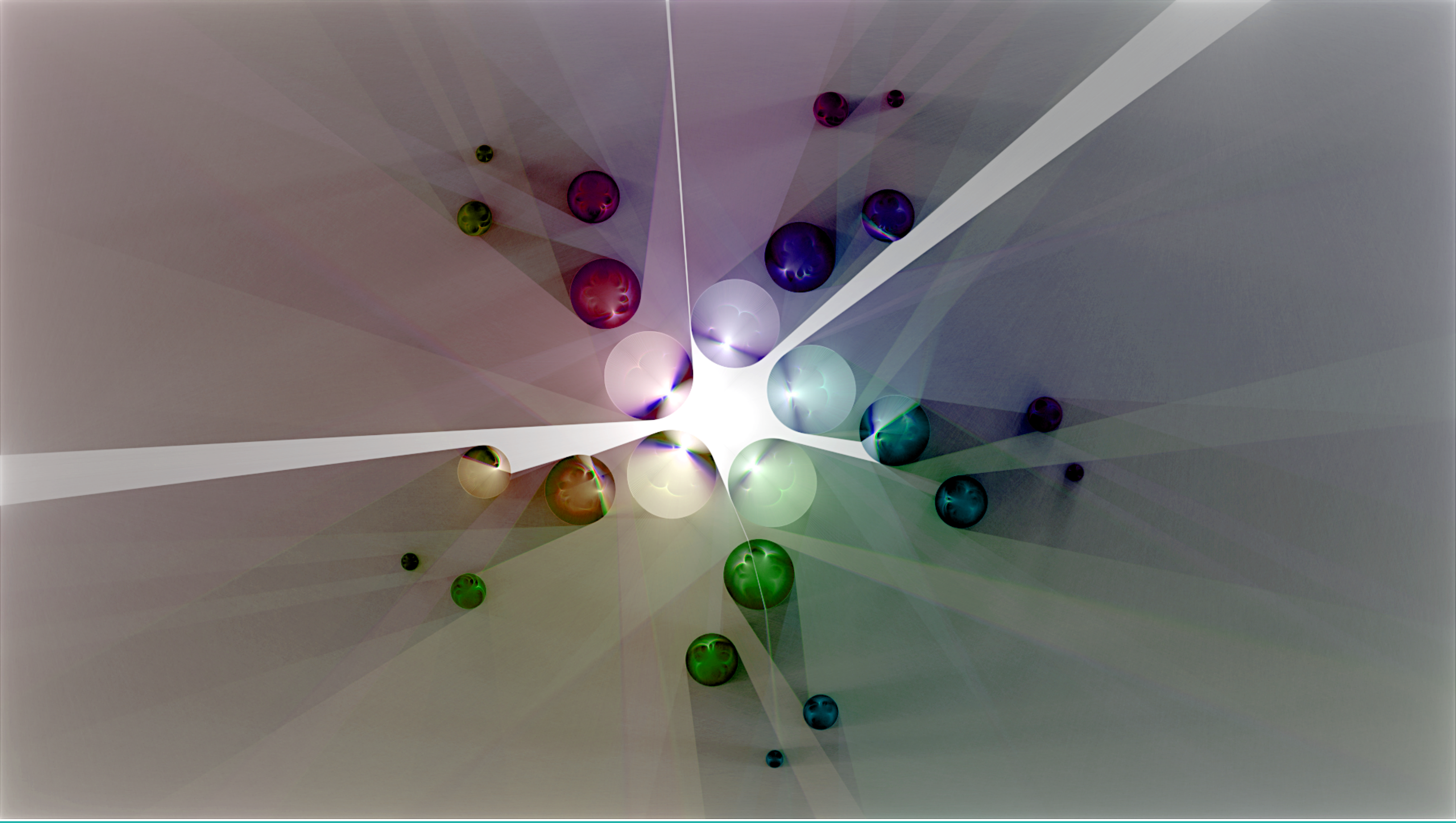


Introduction to Computer Science

Fundamental concepts in CS and how to use them to solve problems

Author: Francisco J. Estrada

Institute: University of Toronto at Scarborough

Date: Apr. 01, 2024

Version: 2.0

Contents

Chapter 1 Introduction

1.1 What this book is about

The goal of this book is to help you explore a few of the most interesting and useful ideas in computer science. These ideas are found in one form or another in the majority of areas of application, research, and study that comprise computer science, and they constitute a powerful toolbox for solving problems in a wide range of fields. The book will introduce these ideas by exploring the kinds of problems they can be used to solve, and will show how they are connected and form a strong foundation for the serious study of advanced computer science topics.

The book is not a programming book, or a book about coding, though we will be doing plenty of that as a necessary component of fully understanding the different ideas and tools being covered. Therefore, a certain amount of familiarity with programming concepts is required. In particular, this book is intended for someone who has gone through an introductory programming course at the college level. No specific prior language is assumed, but familiarity with fundamental programming concepts including **variables**, **functions**, **control structures** such as bf conditionals and **loops**; breaking a problem into the steps required to solve it, and implementing those steps in a program properly organized into functions; and testing and debugging a program. For the sake of generality, the book assumes familiarity with Python, but exposure to a different language such as Java or even C will not be a disadvantage in any way.

The focus of the book is on **thoroughly understanding ideas**, on **developing your problem solving ability**, and on **building a set of skills** that will allow you to successfully dive into more advanced computer science material. The examples, exercises, and problems presented in the book have been carefully designed to guide you in developing these skills and habits; however, they can only be as useful as your own effort allows them to be. To make the most of the book, you should take every opportunity presented to you to practice, think through the content, and exercise what you have learned. As you progress through the book, keep the following skills in the back of your mind, and take the time and effort needed to develop and strengthen them:

- Understanding a problem in terms of the **task and date** involved, and the **expected outcome** of solving the problem.
- Identifying a suitable approach to the solution, based on understanding the concepts and ideas presented in the book.
- Determining whether your proposed solution is reasonable and efficient, based on understanding the amount of work the computer has to do while carrying it out.
- Making reasonable design choices in implementing your solution based on carefully considering the problem, the intended user, and your own experience.
- Predicting what your solution will do given a specific input, so you can properly design tests intended to ensure your solution works correctly all the time.

We will revisit these skills at different points through the book, and we will build them up slowly but continuously as we progress through the material. So, let us dive in and start our exploration!

1.2 Our programming language

Throughout this book we will be using the **C programming language**. **C**, as it is often simply called, was developed in the early 1970's by Dennis M. Ritchie. Currently, the language is extensively used for all kinds of applications ranging from operating systems and robotics, to sound, image, and video processing, to embedded systems software.

The fact that the language has been in continuous use for over half a century should not lead you to thinking that it is outdated and no longer useful, it is in fact one of the top programming languages in use today. Its extensive use means that there is a high probability you will need to use it at different points through your career as a computer scientist, and it is an excellent language to learn as being comfortable with it opens doors for working in serious applications that require a low-level, fast, and well studied programming language. Some of the properties of **C** that make it a good choice for an introductory course in CS include:

- It is an **imperative** language. This means that the structure of a program, and the types of control structures (such as conditional statements, loops, etc.) will be familiar to you from any introductory programming course.
- It is a **simple** language, it provides precisely the support we need to focus on the concepts and algorithms that comprise our course, without having to become distracted by the richness of features, syntax, and programming constructs available in more recent languages.
- It will require you to **know in detail what each line of your program is doing**. Programming in **C** will help you strengthen your ability to fully understand what your code is doing at each step.
- It is a skill that will be required of you in future courses. Whether you are in CS, Math, Stats, or a different discipline altogether, the ability to think carefully and in detail about what your code is doing will help you make the most of courses where programming is an essential component in understanding an exciting topic. If you are in CS, knowledge of **C** will be required for courses such as: Operating Systems, Embedded Systems, Computer Graphics, Computer Security, Computer Networks, Programming on the Web, Programming Languages, and Compilers.
- The memory model used by **C** is very simple, and accurately describes how computer storage works at a low level. It maps directly to how memory is accessed by the computer's processor, and how data and code are processed once a piece of code is translated to the CPU's machine language. Together with what you will learn in a Computer Organization, it will allow you to fully understand how a computer works, and how it can execute a program.

Working through the concepts in this course using the **C** language as your tool will help you develop a thorough understanding of exactly what is going on inside a computer as it goes about solving a problem that is interesting to us. This is possibly the most important habit you can develop in terms of building software. As a computer scientist, you must fully understand what the computer is doing, you must be able to predict what a program you implemented is going to do, you have to develop the ability to detect (by coming up with carefully thought out tests) when your program isn't doing what it should, and you need to be able to find and correct any bugs that may exist in your program before allowing it to be used by others.

All of these things are only possible if you fully understand what the computer is doing at every step of a process or computation. In this book you will find very little (if anything) is hidden from you. We don't use powerful libraries (though many of them are available in **C**) to help us do work - because unless we thoroughly study what the library does we would not know exactly what our program is doing. We don't avoid studying and understanding technical details of how the computer is processing information, and how your program is storing, moving around, and modifying the input it is working on - because then we would not fully understand how the program achieves its work. And we do not take shortcuts in order to get to where we are going faster (though many such shortcuts exist, for instance, we could use automated code-writing tools, or modify existing programs someone else wrote) - because then we would not develop our own ability to think through a problem from beginning to end, and then develop the program that solves it.

Everything you need in order to understand the key ideas this book contains will be discussed within these pages. This means our very first step must be to learn about our programming language, and become familiar with the tool that will allow us to explore the interesting ideas the book is about.

1.3 Structure of a program in C

First things first. A **C** program is just **a text file that has the extension .c**, that means you can use any text editor you like and are familiar with to write and edit your programs. While there is a wide range of tools and IDEs (Integrated Development Environments) you can use to do this, for the purpose of this book and its contents, it would be best if you used a simple text editor with syntax highlighting such as **Notepad+**. The reason for this is that IDEs are intended to be time-saving tools for software development, and because of that they do a lot of work for the developer transparently (without telling you about it), producing code we haven't carefully thought through ourselves. Since we are starting our study of **C**, we want to learn everything regarding how a program is written, compiled, and run. So our best tool at this point is a basic text editor with syntax highlighting.

Secondly, **C** is a **compiled** language. If you're familiar with Python, then you are used to being able to type Python commands within an interactive terminal, and having Python immediately carry out those commands and provide any relevant output. If you write a script in Python, you can directly import functions from the script, and you can run code from the script without any additional work. However, **C** is not like that. The text file that contains the **C** program can not be run by your computer, there is no interactive terminal for **C** in which you can type commands that are immediately executed. Instead, in order to run a program you **must carry out these 3 steps:**

- Edit your program in the text editor, and save it as a file with the extension .c
- Compile your program using a **C** language compiler such as **gcc** (we'll explain this shortly)
- Run your program

We will get plenty of practice with the above process during the rest of this chapter. But before we can get there, we first need to learn what the structure and parts of a **C** program are, as well as the essential building blocks of every program we will need to write.

The basic structure of a **C** program will not be surprising if you have written any programs before, whether in Python, Java, or any other programming language:

- A program is split into functions, each function carries out a specific task related to implementing a given algorithm.
- All **C** programs start at a function called **main()**. This function is in charge of using the rest of the code in the program to carry out the specified algorithm.
- **C** programs can include and use code from libraries (similar to Python modules), which provide a wide range of functionality.
- Blocks of code that belong together are grouped by using delimiters, in **C** these are curly braces **{}**.
- We can add comment blocks to document and explain our code.

1.3.1 A word about data

We write programs for the purpose of manipulating some form of information, which we usually call **data**. This can be anything, from simple lists of number or lines of text, to complete movies that include video tracks, audio channels, subtitles, and other information. All of the data must eventually be stored in the computer's working memory in order for our program to be able to access it, and process it in whatever way is required.

The important thing to keep in mind when you are working with data in **C**, is that for every different data item our program needs to work with, we **must let the computer know what kind of information it represents**. In programming terminology, each data item has an associated **data type**. Data we will work with will usually be stored in **variables**. There are two unbreakable rules about using variables in **C** that you must learn and remember:

1) A Variable has to be declared before it can be used. This means, your program must explicitly state **the name of the variable**, and **the variable's data type**.

2) After a variable has been declared, you can not change its name or its data type. That is, if your program said it is going to create a variable called **x**, and this variable will be used to store **integer** values, then this variable will always be called **x** and can only ever store integers.

There are good technical reasons why the two rules above are needed, and we will get into some of those later on. For now, just remember these two rules.

The fundamental data types supported by **C** are:

- **int** - An integer number. Keep in mind there is a limit to how big this number can be (**C** can't handle infinitely large integers)
- **float/double** - Floating point number, used to store real valued quantities (e.g. **3.14159265**)
- **char** - A single character such as **'A'**, or **'#'**. The list of characters that a **char** can store can be found in the **ASCII table**)
- **void** - This indicates **no data type**, and can't be used for variables, but we will see later it has important uses in our programs

As you can see there aren't many data types. Part of our course will consist of learning how to use these basic data types as if they were Lego building blocks, and that with the basic types shown above we can build powerful data containers such as the lists and dictionaries you may have used in Python.

1.3.2 Overall structure of a C program

At the coarsest level, a **C** program looks like the listing below. Do not worry about the syntax right now, it will be covered in detail soon, instead, pay attention to the parts that comprise the program and see how they are organized.

```
// This is an example program. The '//' indicates a comment, anything
//   found after, all the way to the end of the same line of text, is ignored.

/*
    You can also write entire comment blocks without needing a '//' for
    each line if you start with a '/*'
    Anything that follows is innored until the end of the comment
    block, which is indicated by
*/

// Now for the structure of the program:

// Part 1)
// At the top, we list the libraries our code will use.
// To import a library, we use the '#include<>' statement, similar
// to a Python import.

#include<stdlib.h>  // This is the standard C library
#include<stdio.h>  // This is the standard input/output library

// Part 2)
// After the libraries, we will find the code that comprises the
// program, organized into functions.
// Notice that curly braces {} are used to indicate where things
// begin and end. This applies to functions, loops, conditional
// statements, and other code structures

void a_Function_In_C(int x,int y,int z)
{
  int w;            // Variable declarations should be at the top of a function

  // program statements, as many as needed, including calls to other
  // functions

  another_Function(); // Function calls are pretty much done the same way as in Python
}

// There will be as many functions as needed for the program to carry
// out its work (and we'd expect to see code for 'anotherFunction()' which
// our program seems to be using.

// Part 3)
// The main() function. Every complete program must have it, and
// the program always starts from the code in main()

int main(void)
{

  int x;               // An integer variable called x
  double y;            // A floating point number called y
  char one_character;  // A single character
  program_statement_1; // Every program statement in C ends with ';'
  program_statement_2; // if you forget the ';', you get an error!

  a_Function_In_C(1,2,3); // a sample function call!
                           // There will be more functions and code
                           // in a typical program!
}
```

As you can see there's nothing very surprising. Python programs are organized in much the same way, and while the syntax looks different, the thought process that goes into organizing your program is the same. We will soon get into all the important details, but first let's implement and run our first **C** program. This will also get you started with practicing the **3-step process** we will be using every time we write and run a program throughout the entire book.

1.4 Saying Hello!

A tradition that goes back to 1978, is that your first program in **C** should do nothing more than say **"hello, world!"**. This comes to us courtesy of Brian Kernighan, a Canadian computer scientist who co-wrote the first, and still standard, book on **C** programming: *"The* **C** *Programming Language"*. Let's have a go at it, and as we do so we will review the 3-step process that we have to follow every time we want to edit and run a **C** program.

1.4.1 Write the program

Type-in the program below (or copy and paste it from here!) into a text file called **HelloWorld.c**, don't forget to **save** the program once you're done, and **make sure to note where the program was saved**, you will need that information in a moment.

```
/*
  Hello World!
  Welcome to programming in C
*/

#include<stdio.h>

int main()
{
  printf("hello, world!\n");
  return 0;
}
```

Note

The **printf()** function is part of the standard input/output library, and prints formatted information to the terminal. It has a large number of options that we will study later, but for now, suffice it to say that the string to be printed goes between the quotes, and the **'\n'** at the end is the newline character and tells the function that it should go to the next line after printing (without it, if you have another print statement, the output of the second one will be printed immediately after the first one, on the same line).

1.4.2 Compile your code

Unlike Python, **C** doesn't have an interactive mode, and the text inside your program file can not be run as-is. Instead, we need to use a program called a compiler to generate an executable version of our program. The compiler's job is to read your code, check it for syntax, make sure the code follows all rules of the **C** programming language, and then translate the text in the program to machine instructions in an executable format. You can then run the executable.

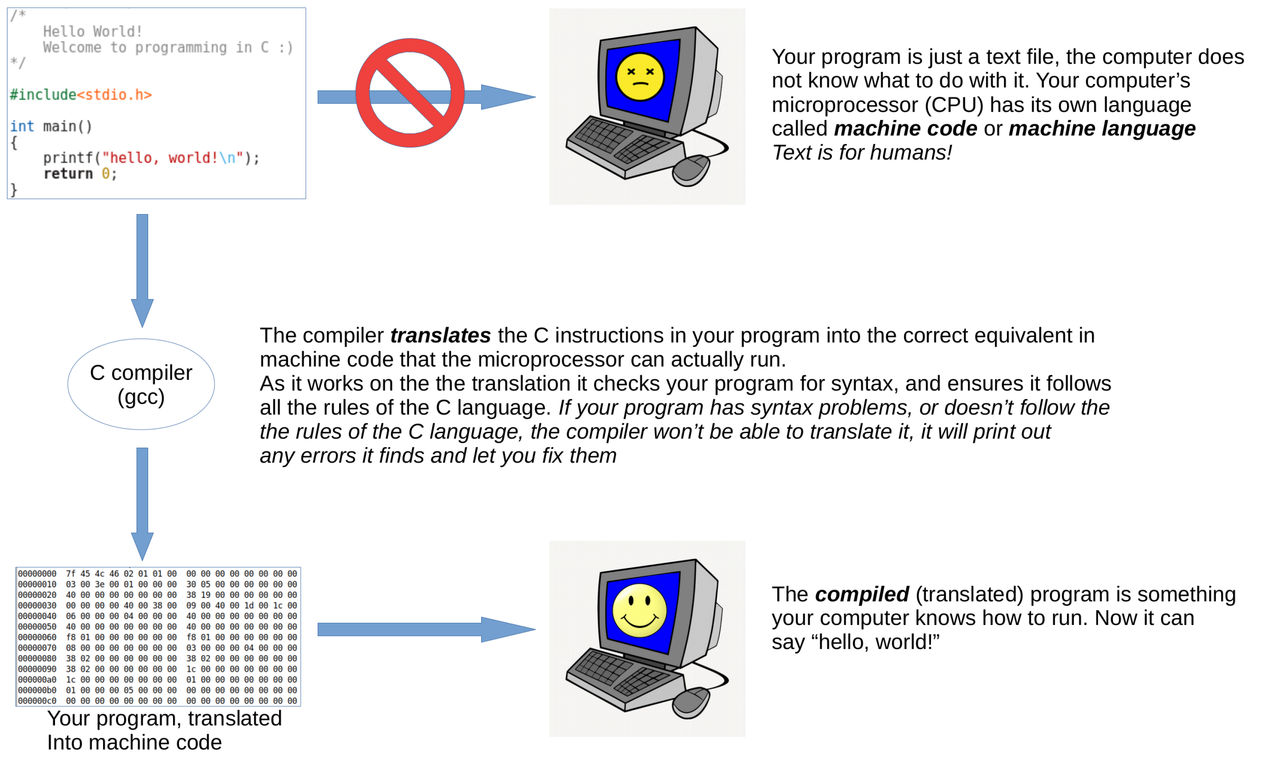
The compiler will report any syntax or structural errors in your code (much like Python will report syntax problems when you load a program), but even if your program compiles without errors, that doesn't mean the program is actually correct in terms of the work it has to do. Errors can happen while the program is running due to mistakes in the program logic or incorrect implementation of an algorithm. Such errors can not be detected by the compiler, they will require careful testing to identify and fix. This will be a topic of serious discussion later on.

To compile your code:

- Open a terminal – depending on what computer you have, and its **operating system** (i.e. Mac, Windows, or Linux), the process to do this will be slightly different. If you don't know how to do it, you can easily find out by Googling **'How can I open a terminal in [my O/S]'**.
- Using the **'cd'** command to change your directory to the location in your computer where you saved your program. If you've never used a terminal before, this will be new to you, not to worry, you can easily find out how to do this by asking Google for instructions for your operating system.
- Once you have your terminal in the directory that contains your code (you can type **dir** to list the files in that directory and check your program is there), you can call the compiler to create an executable for your program.

Note

What are we doing here? and why is this needed? The program we wrote and saved in a text file is intended to be read by humans. The **C** language is meant to be read and understood by software developers. Your computer's microprocessor, on the other hand, does not understand **C** and it doesn't know what to do with a text file. Your computer's microprocessor has its own **very very simple** language called **machine code** or **machine language**. Unlike **C**, or Python, this language can only do very very simple things - such as moving one piece of information from one place to another, or performing some computation on pieces of data, or comparing data stored in memory. Because it's so simple, it's hard for anyone to write long, complicated programs directly using machine language. So instead, we use more powerful programming languages that are easier to read and understand, and that provide lots of features that machine language does not have. But this creates a problem: **we now need to translate our programs from whatever language they are written in, into machine code**. In the case of **C**, the **C language compiler** performs this task. It takes as input the text file that contains your program, and translates it into the correct sequence of machine language instructions that carry out all the **C** instructions you wrote in your code. This process is illustrated in Figure 1.1.



```
/*
    Hello World!
    Welcome to programming in C :)
*/

#include<stdio.h>

int main()
{
    printf("hello, world!\n");
    return 0;
}
```

Your program is just a text file, the computer does not know what to do with it. Your computer's microprocessor (CPU) has its own language called ***machine code*** or ***machine language***
Text is for humans!

The compiler ***translates*** the C instructions in your program into the correct equivalent in machine code that the microprocessor can actually run.
As it works on the the translation it checks your program for syntax, and ensures it follows all the rules of the C language. *If your program has syntax problems, or doesn't follow the the rules of the C language, the compiler won't be able to translate it, it will print out any errors it finds and let you fix them*

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00
00000010  03 00 3e 00 01 00 00 00  30 05 00 00 00 00 00 00
00000020  40 00 00 00 00 00 00 00  38 19 00 00 00 00 00 00
00000030  00 00 00 00 40 00 38 00  09 00 40 00 1d 00 1c 00
00000040  06 00 00 00 04 00 00 00  40 00 00 00 00 00 00 00
00000050  40 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00
00000060  f8 01 00 00 00 00 00 00  f8 01 00 00 00 00 00 00
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00
00000080  38 02 00 00 00 00 00 00  38 02 00 00 00 00 00 00
00000090  38 02 00 00 00 00 00 00  1c 00 00 00 00 00 00 00
000000a0  1c 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00
000000b0  01 00 00 00 05 00 00 00  00 00 00 00 00 00 00 00
000000c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

Your program, translated
Into machine code

The ***compiled*** (translated) program is something your computer knows how to run. Now it can say "hello, world!"

Figure 1.1: Compiling a **C** program creates an **executable** program. This is just a sequence of **machine code** instructions that carry out the process described in your **C** program.

We will be using the **gcc compiler** (the name comes from GNU Compiler Collection). It is a free, open-source compiler that supports **C**, **C++**, and other variants, and is widely accepted as the standard compiler for working with **C**. Your computer may not have a compiler installed. To check if you have the compiler installed, simply open a terminal, and type **gcc –version** then press *ENTER*.

If the compiler is properly installed, you will see something like this:

```
> gcc --version

gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

The **specific version of the compiler doesn't matter**. The only thing that matters is that if the compiler is not installed, you will instead receive an error message (which will be different depending on your computer and operating system). If you do not have the **gcc** compiler installed in your computer, the first step is to install it. Once more, for this task Google is your friend. Ask it to give you a tutorial on how to install **gcc** on your computer (and provide it with information regarding your computer model, and the operating system it runs). You can also find detailed 'how to' videos on how to install gcc for Mac, Windows, or Linux on YouTube. Or, if you want to try the bleeding age of technology, try asking your favorite A.I. Large Language Model (e.g. ChatGPT, Gemini, etc.) to guide you step by step in installing the compiler on your machine.

You will know you have successfully installed the compiler when you can perform the check described above and receive the corresponding version information and copyright notice for your compiler.

Once you have gcc installed, you can compile your code by typing (without the '>' - this is just the terminal prompt and may be different on your computer):

```
> gcc HelloWorld.c
```

This will produce an executable program called **a.exe** in Windows, or **a.out** in Linux and Mac. You can run this from the terminal by typing

```
> a.exe
```

(or **a.out** in Linux or Mac)

If you want the executable file to have a specific name, you can do so by adding the **-o** option as shown below:

```
> gcc HelloWorld.c -o HelloWorld
```

Which will produce **HelloWorld.exe** on Windows, or simply **HelloWorld** on Linux and Mac.

1.5 Your first exercise!

You now know enough to try a little exercise, just to be sure you've properly understood all the material discussed above. You will write your own version of the hello world program. Open a new file, and call it **MyHelloWorld.c**. Make the program print out information about yourself, similar to what is shown below. The specific information you choose to print does not matter, writing the program, and successfully getting it to compile and run is the point of the exercise.

Here's an example of what the output for your program might look like:

```
Hello! My name is: Paco
I am a student in: Computer Science
My favourite colour is: Blue
My favourite fruit is: Pomegranate

I am going to learn C! :)
```

Note that there is an empty line in the output. Write, compile and run your code. Chances are, the first time you try this out you will run into all kinds of compiler messages. Here's the most important thing about learning to deal with compiler errors:

Do not stress out – they happen often and you can fix them

1.6 How to deal with compiler errors

Getting a long list of compiler error messages is a frustrating experience. You will have this experience at some point, and however frustrating it can be, you will be fine if you can:

Take a deep breath

Remind yourself that this is something every single person who has written programs in **C** has experienced.

Know that there is no error message that you can't fix with a bit of help from the many, many programmers who had the same error before you did.

The compiler goes through your code in order, from top to bottom. So errors will be reported in this order. Here is a solid process you can use to find and fix any errors the compiler may have reported in your program:

- Starting at the top of the error report (i.e. with the error that is closest to the top), **carefully read the error message** and go to the line in your program that the compiler says is where the error was detected. Carefully check for typos and for simple syntax errors (e.g. missing parentheses, missing semicolons, etc.). You may need to check a couple of lines above the one the compiler complained about.
- If there are no visible syntax problems, think through what the error message states, and consider how it relates to the code in the line reported by the compiler as well as the couple lines above. It is entirely possible you have never encountered this error before. If the error message is not clear, look for help. Here's a pretty good reference for common error messages as well as their explanation: `https://aop.cs.cornell.edu/styled-4/index.html`.
- If checking a reference like the above does not help you figure out what the error means, it's time to go to Google. Copy/paste the error message into the search box **but do not include any additional information** such as the program's name, or line numbers. Google doesn't know your code, but it probably has seen the same error message in thousands of posts before.
- Carefully review the first few hits Google comes up with, chances are you will find an explanation of the problem, and suggestions on how to fix it. The internet is full of helpful information regarding pretty much any aspect of programming in C, so learn to use it well.
- Done fixing one error? Go on to the next one. Keep going until no errors are left.

Keep a journal of errors you encountered, and what caused them. So you can refer to it in the future and save time and effort. You will in time remember these things without having to visit your journal, but while you're learning, it's a great tool.

Note

The compiler produces two different types of messages you should pay attention to:
An **error** means there is a syntactic or semantic problem with your code. The program you wrote is incorrect by the rules of the **C** language, and it can not be compiled until the error is fixed.
A **warning** means that though your program is technically correct by the rules of the C language, the compiler has noticed something that is likely to get you in trouble when you run your program. The compiler will produce an executable program (the compilation is successful), but running your program may result in unexpected behaviour, your program may have a bug, or it may be doing something different than what you intended.
You **have no option but to fix errors** because the compiler will not produce an executable program otherwise But you should also make it a habit to **fix all the warnings** as well - ignoring warnings is a pretty good way to get into trouble with bugs and unexpected behaviour.
This will become more and more important as we move on to more interesting ideas, and start implementing more complex programs. You should start developing the habit of resolving any compiler messages from the start.

1.7 A few notes on printf()

The **printf()** function will be with us all through the book. We will use it, of course, to output information and results our program is producing. And we will use it extensively to test and debug what our program is doing. You should become familiar with how it works. The standard format for **printf()** is as shown below:

```
printf("...formatting string...", variables, to, print, separated, by, commas);
```

The formatting string specifies what **printf()** is going to do with the list of variables it is printing, it tells **printf()** the data type of the variables it is receiving, so that they are printed with the correct format. Formatting options are specified using the **'%'** character as follows:

```
%d - Prints a decimal (integer) number
%f - Prints a floating point number
%c - Prints a single character
%s - Prints a string
%g - Prints a floating point number using scientific notation
%p - Prints a pointer (this will be formated as a hexadecimal value)
```

There are many more formatting options, and you can specify things such as the number of digits to print for the integer and fractional parts of a floating point number. If you are looking for a way to give your program's output a specific format, look at the on-line documentation for **printf()**. Chances are the function does what you need, you only need to find the correct format specifier.

Besides these format specifiers for variables, the **printf()** function also makes use of certain control sequences to print special characters. These are identified by the \ character. Common control sequences include:

```
\n - New line (go to the next line and print there)
\t - Print a 'tab'
```

```
    \\ - Print a '\' symbol (otherwise the '\' is taken to be part of a control sequence)
    %% - Print a '%' symbol (otherwise the '%' is taken to be part of a format specifier)
```

Finally, note that in **C**, strings are delimited by double quotes **"...string..."**, whereas individual characters are delimited by single quotes, as in **'C'**.

Putting this all together, we can understand what a given printf() statement does:

```
printf("My variables are: %d, %f, %s \n", my_int, my_float, my_string);
```

The result of the above will be to print a line that looks like

```
My variables are: 10, 3.14159265, Hello World
```

And at the end of the string the **'\n'** sequence moves to the next line, so anything we print after will be output to the next row of text in our terminal. Of course, the data type for your variables must match the format specified in printf(), otherwise you will receive a compiler warning (remember we said above that this likely indicates something is not quite right with your program!).

```
#include<stdio.h>
int main()
{
    float pi;
    pi=3.14159265;
    printf("Printing an int, %d\n",pi);
}
```

Compiling the above results in

```

> gcc printf_test.c
printf_test.c: In function 'main':
printf_test.c:6:12: warning: format '%d' expects argument of type 'int', but argument 2 has
    type 'double' [-Wformat=]
     printf("Printing an int, %d\n",pi);
            ^
> a.out
Printing an int, 190300824
```

Obviously, this is not what we would expect. Hopefully this gives you a very clear idea why you must resolve all compiler warnings. Yes, the compiler has compiled your code and produced a **working executable**, but the resulting program doesn't do what you intended.

✍ **Exercise 1.1**

Write a little program that initializes a variable **pi** to **3.14159265**, then prints out **pi** as an **integer** (should print 3), and as a **floating point** number with increasing fractional part lengths. i.e., the output of your program should look like:

```
  3
  3.1
  3.14
  3.141
  3.1415
  3.14159
  3.141592
```

```
3.1415926
3.14159265
```

1.8 The fundamental C control structures

We are almost ready to jump into the details of how to implement algorithms using **C**. But first, we should have a look at how the control structures and loops you learned to use in Python are written and used in **C**. As you will find out, the concepts and the way you use these structures is identical, the only thing that changes is the syntax of how it looks in **C**. Happily, you can always look up the syntax if you don't remember it at some point, and it will become familiar and comfortable to you as we go along with the book.

1.8.1 Conditional Statements

Conditional statements are quite similar to what you're familiar with from working with Python. We use **if** statements to evaluate expressions and execute code that depends on the results. The structure of **if...else** statements in **C** is as follows:

```
if (condition)   // - The condition must be within parentheses
{
     // Code to be executed if condition is true
}
else
{
    // Code to be executed if condition is false
}
```

Of course, you can use nested **if...else** statements.

```

  if (condition1)
  {
      // Some code
  }
  else if {condition2}
       {
          // Some alternate code
       }
       else if (condition3)
            {
                  // Yet another possibility
            }
            else if ...   // and we can keep going
```

Keep in mind that long sequences of **if ... else if ... else if ...** produce code that is harder to read and debug, so try to avoid having a long sequence of statements of this type. It is always a good idea to spend a bit of extra time thinking through what is the smallest, correct set of conditions your program needs to check, and to remove any redundant checking.

Valid logical statements for the **condition** component can involve any existing variables, constant values (such as **5**, or **'p'**, or **3.1415**), and combinations thereof, as well as make use of any combination of valid comparison operators. Common conditional operators are shown below:

```
a==b       True if a equals b
a>b        True if a is strictly greater than b
a<b        True if a is strictly lesser than b
a>=b       True if a is greater than or equal to b
a<=b       True if a is lesser than or equal to b
a!=b       True if a is not equal to b
```

In addition to the above, we can use logical operators to form conditional statements that evaluate multiple relationships between variables. Common logical operators used in if statements include:

```
||      Logical or (this is a double vertical bar)
&&      Logical and (this is a double ampersand)
!       Logical not
```

Example 1.1

```
if ( (a>b && c==d) || e!=f)
{
  // Run this code!
}
else
{
    // Run this code instead!
}
```

Be careful with the way you use operators. A common mistake is to use a single **'&'** or a single **'|'** instead of the double symbol. The single **'&'** and single **'|'** perform an arithmetic **and**, and an arithmetic **or** respectively (i.e. they are doing computation, not evaluating logical values), the result is a binary number instead of a true/false value.

1.8.2 For loops

Loops are a fundamental programming construct, in **C** they come in two flavours: **for loops** and **while loops**. **For** loops use a **counter variable** to keep track of how many times the loop has been carried out. Often, the counter variable is just an integer that is increasing every time we go through the loop until the desired number of iterations has been performed.

For loops have a simple syntax involving four distinct but necessary components:

```
for ( (1) Initial value of the counter ; (2) Condition that causes the loop to continue ; (3)
     increment for the counter )
{
      (4) Body of the for loop - the instructions to be repeated
          The loop itself is contained within curly braces, these
          are indented so that they align with the 'for' keyword
}
```

Example 1.2

```
#include<stdio.h>

int main()
{
    int i;

    for (i=0; i<10; i=i+1)       // Notice there is no ';' after the ')'. This is important!
    {
       printf("%d\n",i);
    }
}
```

In the for loop above, the first line, containing the **for** keyword, provides all the information we need to understand how many times this loop will be carried out. It uses an **integer variable** called **'i'** as a counter. When the loop begins, the value of **'i'** is set to zero (this is part (1) of the declaration). The loop will be executed **as long as 'i' is less than 10** (part (2) of the declaration), and at the end of each iteration, the counter will be incremented by 1 (part (3) of the declaration). The body of the loop (part (4) of the declaration) is simply a **printf()** statement, but a loop can contain any number of lines of program code.

Question: What is the output of the program above?

1.8.3 Variations on for loops

The **C** programming language is very flexible. This is both useful and also dangerous so you have to be careful. In the case of for loops, you have a lot of flexibility in how the loop will work. You can use different data types as counter variables, which means your counter can be either positive or negative, integer or floating point, or even a character type variable.

The increment itself can be anything (as long as it can be applied to the corresponding counter variable). You can even (intentionally) create endless loops. And you can **change the value of the counter variable inside the loop**. This could happen as a result of operations or conditional statements that depend on what is happening within the loop itself. This is not good programming practice, and you should not do it - the important point is this: **C will expect you to know what you are doing and to stay out of trouble**. If you ask the language to do something, it will do it. It's up to you to learn what is good and reasonable, and what is not.

We will help you along the way by providing guidance and examples of good programming practice. But nothing replaces experience. Practice, try things out, and learn as much as you can from seeing what happens!

Here are a couple examples of the variety of loops you can create in **C**:

Example 1.3

```
      float angle;
      float pi;

      pi=3.14159265;

      for (angle=0.0; angle<2.0*pi; angle=angle+.01)
      {
         printf("%f\n",sin(angle));
      }
```

The code above will print the sine of angles between **0** and **2*pi** at intervals of **.01** radians.

Example 1.4

```
     int i;
```

```
    for (i=100; i>=0 ; i=i-3)
    {
       printf("Counting down, we have %d left!\n",i);
    }
```

The code above counts down from **100** in steps of **3**.

Example 1.5

```
    int i,j; // We can declare variables of the same type in one
             // line, separated by commas.

    for (i=0; i<10 ; i=i+1)
    {
       for (j=0; j<i; j=j+1)
    {
        printf("%d, ",j);
       }
       printf("\n");
    }
```

A nested loop where the length of the loop on **j** depends on the value of **i**.

Question: What is the output of the nested for loops above?

1.8.4 While loops

While loops in **C** are very similar to while loops in Python:

```
 while ( condition )
 {
   //   body of the loop
 }
```

The **condition** is a logic statement that depends on variables accessible to the part of the code where the **while** loop is found. Logic statements evaluate to either true or false. The loop is executed for as long as the condition is true.

Any combination of variables with their corresponding data types can be used, as long as the logic statements being applied to them are valid.

Example 1.6

```

   i=0;
   while (i<25)
   {
       printf("This is a loop that prints an integer %d\n",i);
       i=i+1;
   }
```

A common bug occurs if we **forget to change the value of the variable used in the condition at the top of the loop**. This is easy to do if the loop is long, or if we are used to **for loops** which automatically increment the counter. If you run a program with loops, and it doesn't seem to do anything (it doesn't end), check that you do not have an infinite loop because you forgot to increment a counter variable.

1.8.5 Breaking out of loops

Often enough, we may want to end a loop well before its stopping condition is met. For example, suppose you are searching within a very long text document to determine whether it contains the word **"chameleon"** in it, in **pseudocode** this would look like:

```
while <words remain in the document>
  if current word == "chameleon"
    print out "The word is contained in the document!"
    exit loop
```

Note that we are specifically terminating the loop early – it makes sense, once we have found the word we were looking for there is no sense going over the remaining contents of the document.

It is possible to make a program end a loop at any time by using the **break** statement. This applies both to **for loops** as well as **while loops**.

Example 1.7

```
int i=0;
int x=89211022;

// Use a for loop to check if x is a prime number, by testing whether we can divide x
// by integers from 2 all the way to x-1 and get a remainder of 0

for (i=2; i<x; i++)
{
    if (x%i == 0)        // If x modulo i is zero (dividing x by i leaves nothing left)
    {
        printf("x is not prime, sorry! it is divisible by %d\n",i);
        break;
    }
}
```

The example is chosen to illustrate an important point - in the program above, we find out that **x** is **not prime** in the first iteration of the loop (i=2, and x is divisible by 2). There is no sense in allowing the loop to continue for some 89 million iterations doing useless work, the right thing to do is to end the loop the moment we have found out what we wanted to know.

1.9 Exercises

The best way to practice what you've learned is to exercise (this is true for everything worth learning: Playing a musical instrument, cooking, sports, painting, dancing, etc.). So you should challenge yourself to write a few simple programs that combine **conditionals**, **loops**, and involve using **variables with different data types** as well as printing information using various formatting options available with **printf**().

Here are a few suggestions for exercises that involve the material covered in this chapter:

✍ **Exercise 1.2** Write a program that outputs the letters in the alphabet (from A to Z)

✍ **Exercise 1.3** Modify the program from (1) so that it prints both the upper-case and lower-case version of each letter side by side, e.g.

```
    A   -   a
    B   -   b
    C   -   c
    .
    .
    .
    .
    Z   -   z
```

Hint: You may want to look at the ASCII character table (easily found online), and think of text characters as numbers. C lets you do with **character type** variables pretty much anything you can do with **integer type** variables.

✍ **Exercise 1.4** Write a program that prints out the prime numbers between 2 and 100

✍ **Exercise 1.5** Write a program that prints out all the possible sequences of 3 characters in **'A-Z'**, e.g.

```
    AAA
    AAB
    AAC
    .
    .
    .
    AAZ
    ABA
    ABB
    ABC
    .
    .
    .
    AZZ
    BAA
    BAB
    BAC
    .
    .
    .
    // and so on until you reach
    .
    .
    .
    ZZZ
```

Note that you do not need **strings** - which we haven't learned about yet. This should be done using **character type** variables only.

✍ **Exercise 1.6 A more challenging exercise:** Write a small program that uses a **for loop** to go over the numbers from **1 to 100**, and prints out any that are perfect squares. The output of your program should look like:

```
  4 = 2*2
  9 = 3*3
  16 = 4*4
  .   (there are more perfect squares printed)
  100 = 10*10

  You will need to use for loops, conditional statements, and printf().
```

1.10 Building Programs That Work - Part 1

As we move on through the book, we will work towards developing habits and skills that will help ensure the programs we write work. What this means is that

- They perform the function they are designed to perform, correctly, every time.
- They have been thoroughly and carefully tested to remove as many bugs as possible.
- They have been properly documented so that they can be maintained, improved, and/or expanded by ourselves or others.
- They perform their work efficiently, using an algorithm known to be, or that can be shown to be, not unnecessarily wasteful of computation, storage, or other resources.

We will cover different aspects of building code that works in each of the chapters in the book. For this first chapter, our task is simply to develop good habits with respect to working with a compiler. So, as you work on each of the exercises in the book, or write your own, make sure to spend time exercising these five steps:

- Compile your code from the **terminal**, **not from within an IDE**.
- Look up each of the compiler errors/warnings you haven't seen before.
- Locate the line(s) in your code that produce the specific errors/warnings you're currently observing.
- Figure out what you need to do to fix the corresponding error or warning - this may include doing the work of finding information about the issue online.
- Once you have resolved all errors and warnings (your program compiles cleanly), **run** your program **from the terminal** and verify that it does what you expected.

The more practice you build with these, the better your work process will be, and the less time you will have to spend trying to figure out why a particular program is not working properly.

Chapter 2 The C Memory Model

In this chapter we will learn about how computer memory is organized, and how our programs use it. This is important because at the end of the day, our programs will be working with information stored in the computer's memory. Knowing what is stored where, and how to access and modify it, is essential for understanding what your program is doing.

2.1 Computer memory is like a room full of lockers

If you have been to the change room in a gym, or if you had a locker while you were in high school, you will have noticed the rows of lockers set up there for storage (Fig. 2.1). Row upon row of lockers, some empty, some full, the ones that are full are assigned to a person who has either a key or a combination that allows them to put things into, and take things out of their locker.

Figure 2.1: A locker room. Organized as rows and columns of numbered lockers, each with its own lock or key. *Image: Tiia Monto, Wikimedia Commons, CC-SA 3.0*

All locker rooms share a couple of important properties:

- Lockers are reserved - the owner of the locker has a key or combination that opens the door so only the owner can store things into, or take things out of the locker.
- Lockers are ordered by number in a way that makes it easy to find a specific one.

The process of reserving a locker changes from place to place and is not important to us right now. Neither is the size of the lockers, or the number of them in the locker room. What matters to us is that the properties listed above are very much the same properties of the main memory storage inside a computer. Indeed, at the lowest level, computer memory is just like a very large locker room:

- It consists of numbered boxes where we can store information
- The boxes are ordered by number in ascending order
- Boxes are reserved so that only the program using that box to store information can store or access the information in that box (we will see later what happens when we need to share)
- If you look at the microprocessor under a microscope, you can see the rows and columns of memory lockers (Fig. 2.2)

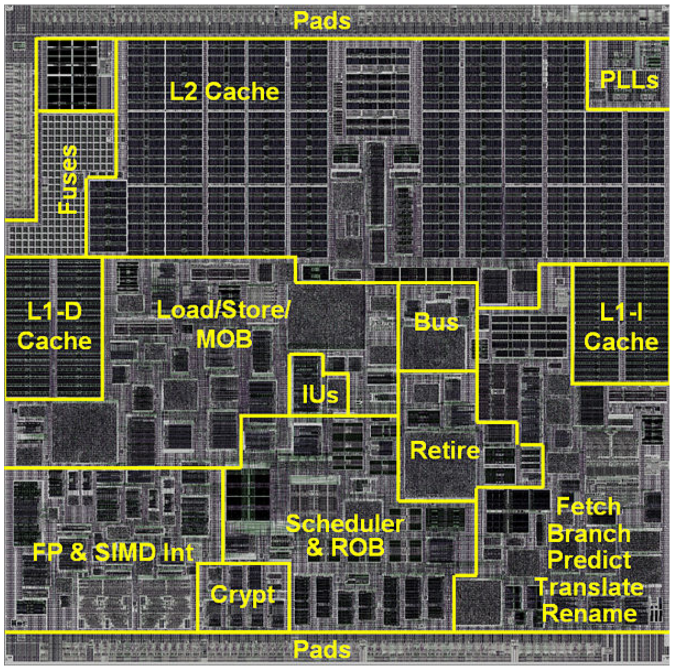


Figure 2.2: This is a micrograph of a CPU. Notice the rows and columns in the regions labeled L1 cache and L2 cache. These comprise this CPUs memory area, and they are indeed organized much like a regular locker room. You can learn all about how this works in any Computer Organization book. *Image: VIA Gallery, Wikimedia Commons, CC-SA 2.0*

For the purpose of this course, we will think of computer memory as nothing more than a large locker room where our programs keep their data. We will see that declaring variables, assigning values to these variables, and moving information between functions is very much the same thing as reserving a set of suitably sized lockers, storing things in them, and moving those things around.

2.2 What happens in memory when we declare a variable in C?

As we saw in the previous Chapter, **C** supports a small number of data types we can use to declare variables. Let's have a look at what happens in memory when we compile and run a program that does something very simple: Declaring a single integer variable and assigning a value to it.

Example 2.1

```
#include<stdio.h>
int main()
{
        int x;
        x=5;
    }
```

Let's picture our computer's memory as that locker room we've been talking about - this could look like the image in Fig. 2.3.

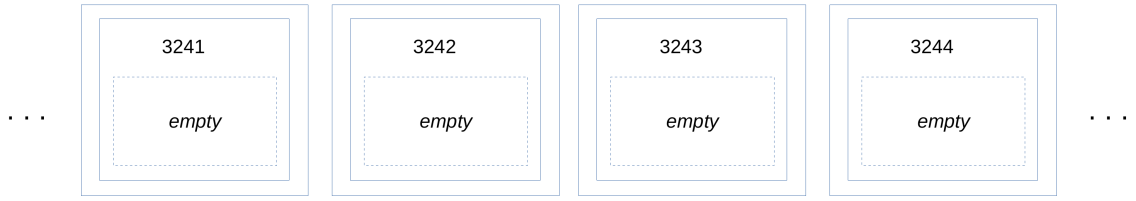


Figure 2.3: A section of the computer's memory where there's **empty** lockers - the specific locker numbers do not matter, they could be anything.

Figure 2.3 shows just a little section of memory, with 4 numbered boxes. The **numbers on them are not important**, just as the locker number you get when you go exercising doesn't matter. All that matters is that they are **empty**, and available for use by a program (in this case, our program!).

Note

What does **empty** mean? - The diagram above shows the four boxes as **empty**. This means that the computer knows these lockers are not reserved for use by any program. However, inside the actual computer memory **there may be junk left over by whatever program last used that box before**. So in practice, you should always assume that an **empty** locker contains junk until you change its contents to something useful. **Be careful with this**, it can cause bugs that are hard to fix because your program looks right – it's just using junk values left over inside a locker you haven't yet put anything meaningful into.

In **C**, **declaring** a variable (first line in Example 2.1):

```
    int x;
```

means that we want to **reserve a locker**, to store **one integer value**, and we want to **tag that locker** with the name **x**. In the computer's memory, a locker suitable for storing one integer value is reserved for our program to use, and tagged with **x**, as shown in Fig. 2.4.

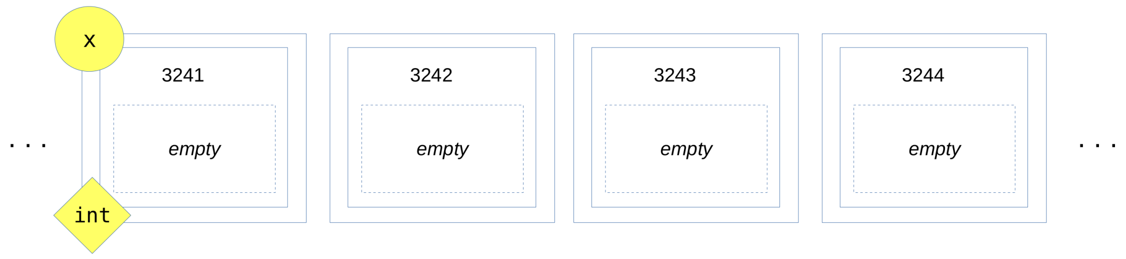


Figure 2.4: The same memory section from Fig. 2.3, after our program has requested space for an **integer** variable called **x**.

Note that:

- **'x' is not inside the box**, the box is still empty, the variable's name **x** is used as a tag so at any point when our program needs **x**, we can easily find the locker that contains the value of that variable
- **x** is also tagged as being of type **int**. Thereafter your **C** program will know what type of data is stored in that box
- Declaring a variable does not initialize it or set it to zero. As you see, the locker remains empty. Be careful with this, some compilers choose to reset variables to zero, and some do not. Or do so only in some cases and not in others. The **safe** practice is to assume your new variable contains **junk** after you initialize it, up until the point where you give it a meaningful value

In our example, locker #3241 is now reserved for our program's use, and it will store whatever values variable **x** will take during the program's execution. The next line in Example 2.1 contains an assignment operation:

```
   x=5;
```

This literally means **go to the box tagged with x, and put a 5 inside it**. The end result is shown in Fig. 2.5.

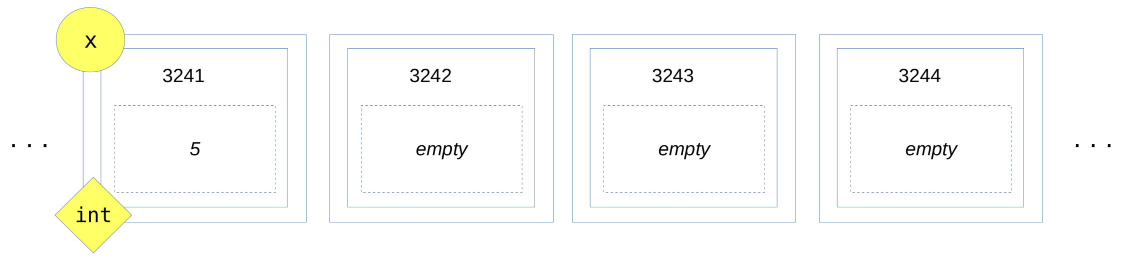


Figure 2.5: The same memory section from Fig. 2.4 after we perform an **assignment**, which puts data (an integer with value 5) inside the locker tagged **x**.

Of course, there could be a much more complicated expression right after the **equals** sign, in which case the expression is evaluated (it must result in a concrete data value) and the result is stored in the corresponding box.

Let's do a more interesting example to practice how all of this works. We will declare several variables, see what the corresponding situation is in memory after the declaration, and summarize the key ideas that explain how variables in **C** work in the computer's memory.

Example 2.2

```
main()
{
   int x;         // Reserve space for an integer variable called x
   float pi;      // Reserve space for a float variable called pi
   char c;        // Reserve space for a variable of type char called c

   x=5;           // Go to the box tagged 'x' and store a 5 there
   pi=3.141592;   // Go to the box tagged 'pi' and store 3.141592 there
   c='C';         // Go to the box tagged 'c' and store the character C in it.

   // The figure illustrates the situation in memory at *this* exact point.
}
```

The situation in memory after the code above is processed will look as shown in Fig. 2.6. We have three variables: **x**, **pi**, **c**, each with their own data type (int, float, and char respectively). We also have assigned values to these variables that make sense for their data type. In the memory model, we see that we have three reserved boxes, each of these corresponds to one of our variables **in the order in which they were declared**, each box is tagged with the variable's name, and the variable's data type, and each box contains the value that we assigned to the corresponding variable.

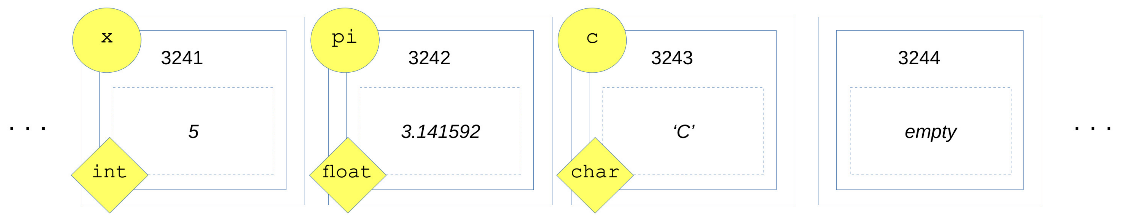


Figure 2.6: This shows the result of processing the code in example 2.2, we have three boxes reserved, and each of them contains a value as requested by the program.

Note

From the examples above, you should take home the following essential facts about variables:

- Variables are just boxes in memory where we can store information
- The variable's name is just a **tag** so the compiler knows which box we want to use
- Each variable **gets its own box**, and each box can have **only one tag**
- **Assignments** put values (data) inside boxes
- All your program (or anyone else's program for that matter) ever does is go into boxes, see or change what is in there, and move data around

✍ **Exercise 2.1**

Draw a diagram like the one above showing what you'd expect would happen in memory if we compile and run the following program:

```
int main()
{
    int x;
    int y;

    x=5;
    y=x;        // What does this do?

    return 0;
}
```

2.3 Functions in C – how information moves about

In your introductory programming course, you learned that we break programs into small **functions** each of which has a specific task to perform, you learned that functions often take input **arguments**, and often **return** some value that is the result of whatever processing the function does. The input arguments and the return value are the foundation of how information moves around inside a program.

Let's now take a look at how we **declare a function** in **C**, and discuss the important properties of how information moves around from one function to another as our program goes about its work. The example below declares a simple function. It has one input argument, and it computes and then **returns** the value corresponding to the square of the input argument.

Example 2.3

```
        int square(int x)      // This is the function declaration
        {
            int s;                  // The body of the function consists of anything contained
            s=x*x;                  // within the curly braces
            return s;
        }
```

First off, let's spend a moment looking at the line that declares the function

```
    int square(int x)
//   ^     ^      ^
//   |     |      \------ Input parameter: an integer variable called 'x'
//   |     \------------- The name of the function
//   \------------------- The function's return type. This one returns an
//                         integer value
```

The declaration has 3 parts always:

- First off, the function's **return type** which tell the compiler the data type of whatever information the function will return. Functions in **C** can only return **a single data value**.
- The function's **name** - you should make sure to give each function a name that appropriately describes what it does.
- The function's **input arguments** - the function can have as many input arguments as needed, separated by commas. For each input argument, we need to provide the **data type**, and a **name**.

Note

Just like with variables, a function's **data type** is fixed. Once defined it can not be changed. The same goes for the argument list, once declared, the function will always expect the correct number of arguments and they all must have the correct data type. If you call a function with the wrong number or type of arguments you can expect the compiler to report an error or warning (depending on the severity of the issue).

After the function's declaration, we find the function's body which contains

- **Declarations for the variables** the function will use. Note that each input argument is **already a variable** usable by the function - for this reason, variables in a function must have a name that is different from any of the input arguments.
- The **function's code**, which can be as short or as long as needed. Good programming practice suggests you should keep functions on the shorter side as much as possible.

- The **return** statement, which **causes the function to send a result back** to whichever part of the program called the function. The function can return any of the variables or arguments it has available, but the value being returned must be of the correct data type.

That is all there is to function declarations in **C**. All the functions you will ever write in this language follow the rules stated above. Of course, a program will often contain many functions, so let's now look at an example where we have two functions, and we will use this example to understand what happens in memory when some part of a program uses a function to do some work.

Example 2.4

```
#include<stdio.h>

int square(int x)
{
    int s;
    s=x*x;
    return s;
}

int main()
{
    int input;
    int result;

    input=7;
    result=square(input);

    // Here we would normally print the result, we'll see how to do that very shortly!

    return 0;              // This is new! we'll explain below
}
```

Let's see what happens in memory when we compile and run the code above:

When the compiler is processing your program, it will note that **main()** declares two integer variables called **input** and **result**. These two variables will be assigned one locker each, next to each other, in the order in which they appear in the program. The compiler will also note that **main() is expected to return an int**. Therefore the compiler will make sure there is a **separate box reserved to store this return value**.

Note

Whatever value a function returns must be stored in its own box, and this box will be reserved right beside the last of the function's variables.

When we **run** the program, the instructions that are executed are the instructions in **main()**. In Example 2.4, the first thing the program does is to store a **7** in **input**. In the memory model, things will look as shown in Fig. 2.7 immediately after the **7** has been stored in the corresponding box.

The next line of the program is important to understand because **all function calls in C work in the same way**:

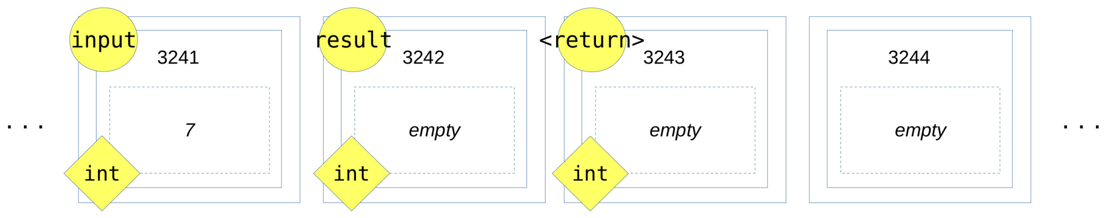


Figure 2.7: A memory model for the program in Example 2.4 up to the line **input=7;**.

```
        result=square(input);
```

It does the following:

1) It reserves memory for the function **square()** to use (notice that space for functions is only reserved at the time when they are called). From the function's declaration the compiler knows it needs space for one input argument **x**, one **local variable** *s*, and the function's **return value** which is of type **int**. This is shown in Fig. 2.8. Notice that the memory locations reserved for **square()** could be anywhere and not necessarily close to the lockers reserved for **main()**.

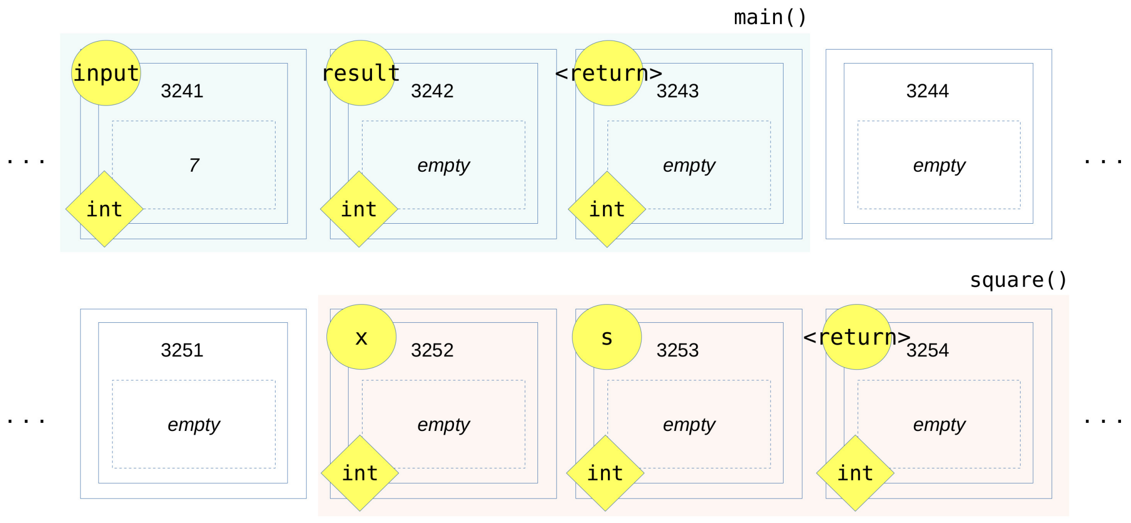


Figure 2.8: Memory model after lockers have been reserved for the **input argument**, **local variables**, and **return value** for function **square()**.

Note

The memory model has been annotated to show clearly which lockers were reserved by **main()** and which were reserved by **square()**. There is an essential property of the memory model that we should make sure to understand at this point: **variables and input arguments that belong to a function can only be accessed by the function that owns them**. This means that **main()** doesn't know anything about **square()**'s variables, their name, type, or where their lockers are. Similarly, **square()** doesn't know anything about **main()**'s variables, their type, or where they are. We say that the variables (and input arguments) declared by a function are **local** because **only the function that owns them can access/read/modify the corresponding boxes**.

Definition 2.1

The **scope** *of a variable is the region within a program's code where this variable can be referenced and manipulated.* ♣

The **scope** of all the **local variables** and **input arguments** declared within a function is the code contained within the curly brackets that delimit the start and end of the function. **There are other possible scopes** for variables in **C**, and we will learn about them as we progress through the course.

2) It **calls the function**, which means: It passes the required arguments to the function and then proceeds with whatever instructions are part of the function itself. In the example, the function was called from **main()** using the **local variable** *input* as the argument to **square()**. So the value of **input** is **copied onto the box reserved for** *x*. This is shown in Fig. 2.9.

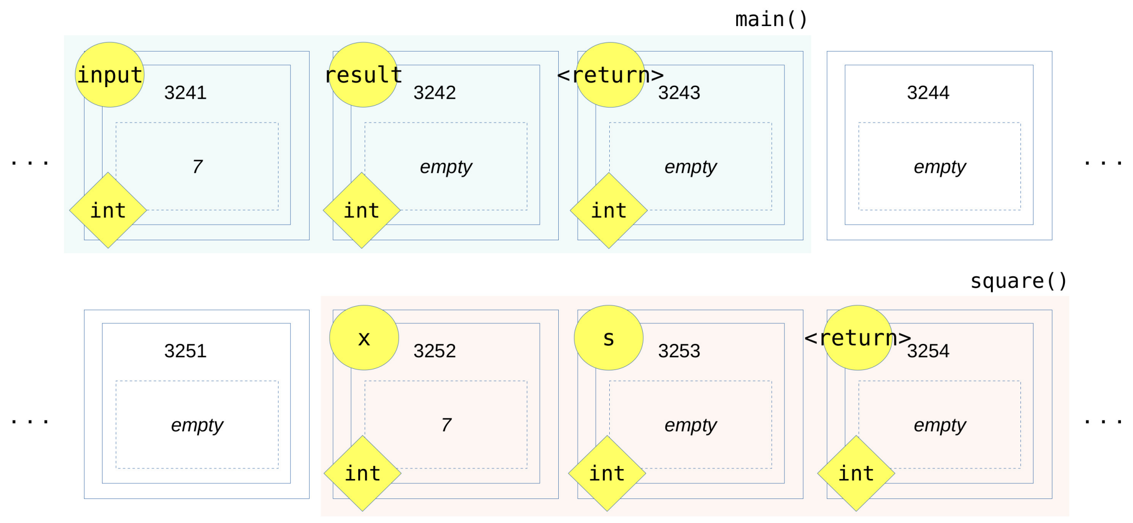


Figure 2.9: Calling the function means **making a copy** of values passed to the function into the corresponding **input arguments**.

3) The program then continues with the instructions that is are part of function **square()**. In this case, it evaluates the square of the input argument and stores it in the **local variable** *s*. The last line in **square()** contains

the **return** statement. In this case, it returns whatever the value of the **local variable** *s* is at that point. **This makes a copy of the value of** *s* **and stores it in the box reserved for the** *return value* **of square()**, as shown in Fig. 2.10.

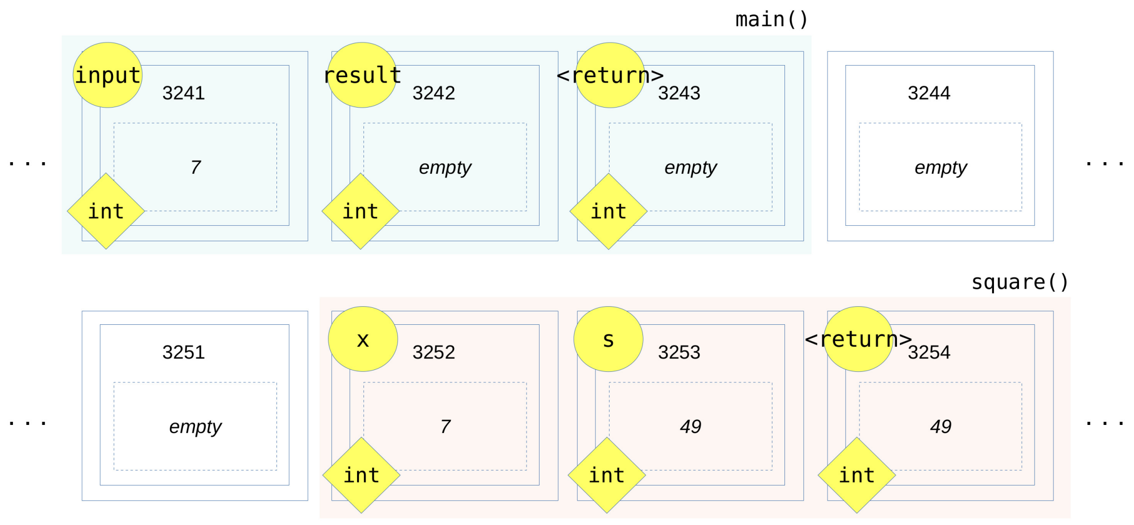


Figure 2.10: Memory model after the instructions in **square()** are carried out, the last line copies the value we want to return into the box reserved for the function's **return value**.

4) The program then returns to **main()** exactly to the line where **square()** was called, and copies the **return value of square()** onto the **local variable** *result*. This is illustrated in Fig. 2.11

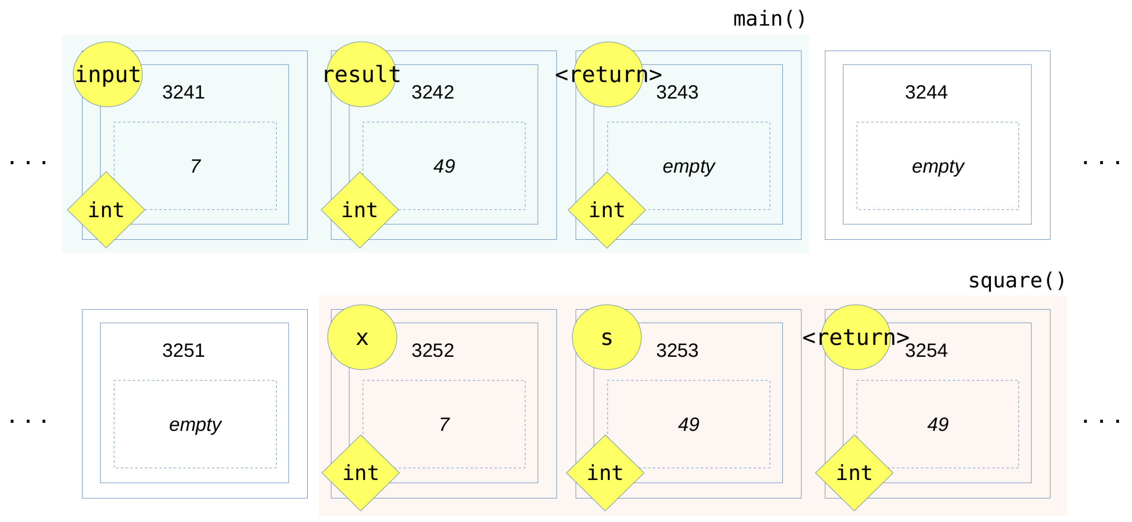


Figure 2.11: When the function **returns**, whatever is stored in its **return value** box will be **copied** into **main()'s local variable** *result*.

5) Because the work of **square()** is complete, the memory reserved for the function is **released**, which means these lockers become available for any other function or program to use for their own work. This is important, **memory reserved for a function is only reserved for as long as the function is doing work**. As soon as the

function returns, its reserved lockers are released. The final state of the memory for our program will look as shown in Fig. 2.12.

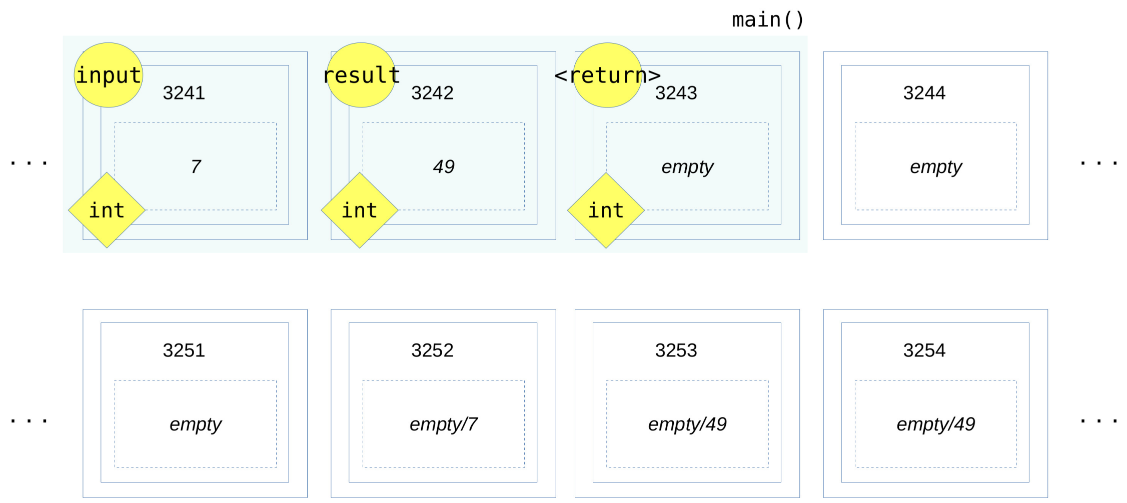


Figure 2.12: Memory model for the program after the function **square()** has returned, its work is done, and memory reserved for it has been released.

Note that **the computer does not clean up** the boxes that were used by **square**. They are marked as **empty**, and they can be reserved by any other program or function, but whatever was left there, is still there. This is why, as discussed previously, we should assume that lockers assigned to our program will contain **junk information** until we put something meaningful there.

After the call to **square()**, the program continues with any instructions left in **main()**, which could, for example, print the result, or there could be more function calls, etc. In the example above there's no more work for **main()** to do, and we find a **return** statement that signals the end of **main**. Why does **main()** have a return value? who gets that return value?.

Note

All programs have to be **started by something/someone**. The simple programs we have written so far are launched by us from a terminal, but the same thing happens when you double click on a program on your computer's desktop, or when another application launches a helper program to do some work. The point is, the program **is called by something outside itself, much the same way functions are called within a program**. So, by **convention**, a **C** program must have a **main()** function that returns an **integer value**. The point of this return value is to provide information to whatever user or application launched the program. Most of the time, the purpose of the return value is to **indicate whether the program completed its work correctly, or conversely, whether there was some error and the program did not finish properly**. The expected return values are defined as:

- **0** indicates the program completed correctly, with no errors
- **any positive integer greater than zero** indicates an error occurred, and the value of the integer can be used as an error code to indicate what the error was. There is no standard for what non-zero return values mean, it is up to the application to provide documentation that explains different possible error codes

2.3.1 Summary

From the examples above and the discussion on how information moves around between different functions in a program, you should make sure to remember the following principles that apply to **every function, and function call in C**.

- Each function will have **its own memory space**. Functions can only access/change information that is stored in boxes reserved for the function, no other part of the program has access to these boxes.
- Because each function has its own lockers, we say that variables and input arguments are **local to the function**.
- For each function, space is reserved for **each input argument** in the order in which they appear in the function declaration, then space is reserved for **each local variable** in the order in which they are declared, and finally a box is reserved for the function's **return value**.
- Multiple functions can have variables or input arguments with the same name, since **they each have their own box** within **the memory space reserved for their function**. There is no confusion about which variable some part of the program is using.
- Information is **passed into the function** by making copies of the **variables passed to the function in the function call** into the corresponding boxes for the function's **input arguments**. This is an essential property of how functions in **C** work. We say that arguments are **passed by value** into **C** functions because we provide the function with a copy of the values it is being passed as arguments.
- Information is **passed back outside the function** by copying the value stored in the function's **return value** box into the **specified local variable** outside of the function.
- A function can only return **a single data item**.
- Once the function's work is done, the space reserved for the function is **released** for use by other functions or programs. This means functions only use memory space when they are **actively doing work**.

✍ **Exercise 2.2** Show in in a memory model diagram what the contents of memory would look like **just before the space reserved for** *square()* is released. Your memory model diagram should clearly show **each function's memory space**, and for each function it should contain **the function's arguments if any**, the **local variables**, and the **return value**. These should be in the order discussed above. Do not forget to **tag each box with the variable's name and data type.**

```
#include<stdio.h>

int square(int x)
{
    x=x*x;
    return x;
}

int main()
{
    int x;
    x=9;
    square(x);

    // What would be the value of main()'s x variable if we printed it out in *this* line?

    return 0;
}
```

Question: What would be the final value for the local variable **x** declared by **main()**?

2.3.2 Input arguments and type conversions

You know by now that all variables, input arguments, and function return values have an associated data type. You can not change the data type at run-time. However, **certain type conversions are possible during the process of passing arguments to a function**, as well as during the **assignment of return values to variables**. Let's see an example:

```
#include<stdio.h>

float square(float v)
{
    // Now this function takes as input a floating point number,
    // and produces a corresponding floating point result.

    return v*v;    // In C, we can do a little bit of
                   // processing in the return statement!
}

int main()
{
    float pi;
    int x,y;
    int result;

    x=5;

    // Notice we will call square() with an int, not a float,
    // and we are assigning the result (which is float) to an int.
    y=square(x);
    printf("The first result is %d\n",y); // What does this print out?

    pi=3.14159265;
    y=square(pi);
    printf("The second result is %d\n",y); // What does this print out?

    return 0;
}
```

Note that we have changed the declaration of the function square(). It now takes as input argument a floating point value **v** and returns a floating point number that is the square of **v**. This time we shortened the function a bit so the calculation happens right at the return statement. Draw for yourself a diagram of what's going on in memory when you call this function.

Here's what happens when we compile and run the little program:

```
>./a.out
The first result is 25
The second result is 9
```

What's going on here?

- The first time we call **square()**, we are passing in **x** which is of type **int**. Because **square()** expects a float as an input parameter, the value of **x** which is **5** is converted into a **floating point** value **5.0**, and then stored in the corresponding input argument **v**.
- This process of converting information from one data type to another is called **type-casting**.
- After **square()** has done its work, the result which is the **floating point** value **25.0** is returned. However, it is being assigned to local variable **y** which is of type **int**. The floating point value **25.0** is type-cast to the integer value **25** and stored in **y** - so the first print statement prints out **25**.
- The second time we call **square()** we are passing in **pi** which is of type **float**. No conversion is necessary, so the value **3.14159265** is copied onto **v** and the square is computed and returned.
- However, the result, **9.869604** is being assigned to local variable **y** which is an **int**. Once more, the floating point value is converted to an integer. **This is done by chopping off the fractional part, the conversion does not round the value to the nearest integer**. The resulting value assigned to **y** is **9**, and this is what the second print statement outputs.

Note

The **conversion**, or **type casting**, is transparent to you but you must always be aware when a type conversion is taking place, as this is a place where bugs can easily creep into your program. In fact, good programming practice would have you make sure type casting only happens when the programmer explicitly intended for it to happen.

There is a consistent set of rules used to type-cast values in C. You can learn how these rules work, but it is a bad idea to write code that assumes you know these rules perfectly – worse, it makes your code hard to debug: Anyone reading the code would have to be an expert to identify and solve problems introduced by unintended type-casting. Here's an example of why this is important.

Example 2.5

```
#include<stdio.h>

int main()
{
    int x,y;
    float result;

    x=5;
    y=2;
    result=x/y;
    printf("The result is: %f\n",result);

    return 0;
}
```

Compiling and running the code above produces:

```
./a.out
The result is: 2.000000
```

The result is not correct despite the fact that **result** was declared as a **float** and should be able to represent the value of **5/2**. However, because the expression **x/y** must be evaluated first, and since both **x** and **y** are **int**, the division carried out is an **integer division** which gives the integer result of **2**. The integer **2** then is **type-cast** into a floating point value **2.0**. **To ensure things work out properly in your code you must explicitly indicate when type conversions should happen, and what data type the conversion should result in.** A correctly written version of the code above would look like this:

```
#include<stdio.h>

int main()
{
    int x,y;
    float result;

    x=5;
    y=2;
    result=(float)x/(float)y;
    printf("The result is: %f\n",result);

    return 0;
}
```

In the program above, the expression **(float)x** indicates we **are requesting the compiler perform a type conversion for us**. In this case, it tells the compiler to convert the value of **x** into a **float** before it's used. Likewise **(float)y** tells the compiler to convert the value of **y** into a **float** before using it. Since the division is now working with two floating point numbers, we should expect the result to come out right:

```
./a.out
The result is: 2.500000
```

The code above is an example of **explicit type casting**. We tell the compiler **when** to perform a type conversion, and **what data type** we want the result to have. It is essential that we do this whenever we are writing code that uses **mixed data types** such as **integers** and **floats** (this is quite common).

Note

You **must be very careful** when performing type conversions to ensure that the result makes sense and is valid. This is an issue because **floating point** variables can store values that are much, much larger (or much, much smaller) than any integer can handle. So for instance, if you tried to type-cast the floating point value **12345678987654321.0**, which has no fractional par, into an integer; you would get **4294967295**. This is clearly wrong! but happens because the original value is too large for an integer to hold. **This is unavoidable** and a property of these two data types. Similarly, any floating point value smaller than **1**, will give you **zero**.

Not only must you perform the type conversions explicitly in your program (as shown above), but you also **have to implement safeguards** to make sure the values that result from the conversion will be valid within the context of your program, and will not cause a bug. To summarize, you should always carry out **explicit type conversion** because

- It will help you keep in mind a complete and clear picture of what your code is doing, what the expected inputs and outputs to computations and functions should be, and ensures that data is being manipulated as you expected at all times.
- It greatly reduces the possibility that you will run into unexpected bugs caused by type conversions being performed without appropriate safeguards. This is no small thing, Fig. 2.13 shows an example of the kind of major fault that can result from not being careful with type conversion.

Figure 2.13: An Ariane-5 rocket self-destructed after takeoff due to software error caused by an incorrect type conversion. The cost of this mistake was over 370 million dollars. *Photo: ARIANE 5 Flight 501 Failure Report by the Inquiry Board (*`http://sunnyday.mit.edu/nasa-class/Ariane5-report.html`*), Public Domain*

Note

So why are data types such a big deal?

All the information in digital computer is stored in the form of bits, so every piece of information you will ever manipulate with a standard digital computer, from integers, to floats, to music videos must be represented, in some way, as a string of ones and zeros.

How the bits are interpreted to represent information depends on what type of information is being stored. Integers and floating point numbers have completely different binary representations. This means that the circuitry inside the CPU that carries out operations with these different types of binary data can not mix and match bit strings for integers with those for floating point numbers.

Therefore, all CPUs have separate instructions for doing arithmetic and logic operations on integers, and on floating point values. You can see that these circuits are even located on different parts of the CPU surface by looking at a micro photograph of the CPUs circuitry in Fig. 2.14.

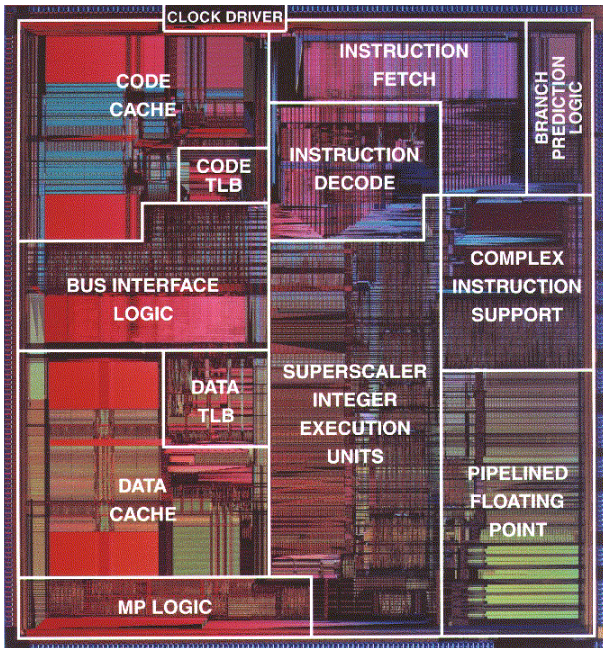


Figure 2.14: Micrograph of a Pentium CPU showing the separate circuits for the integer (labeled Superscalar Integer Execution Units), and floating point (labeled Pipelined Floating Point) arithmetic units. *Photo courtesy of: CPSC 3300 Computer Systems Organization, Clemson University*

2.4 Arrays

In **C**, we can reserve space for more than one data item of a given type. The resulting set of items is called an **array**, and it is the simplest data structure that will allow you to work with sets of data.

Here's how we declare an array in C:

```
#include<stdio.h>
```

```
int main()
{
    int my_array[10]; // This is an array of 10 integer values

    my_array[0]=10; // This is the first entry in the array
    my_array[9]=5;  // This is the last entry in the array

    return 0;
}
```

The syntax should look familiar to you, and you've likely used similar syntax to manipulate lists in Python. Be aware that **arrays are very different and much simpler than Python lists**. The only operations they support are reading and storing values in the different array locations.

In the memory model, the array declaration shown above causes **a group of 10 consecutive lockers, all of them able to store an integer** to be reserved for the program. The **first box, with the lowest locker number** is tagged with the **array name** and **array type**. This is shown in Fig. 2.15.

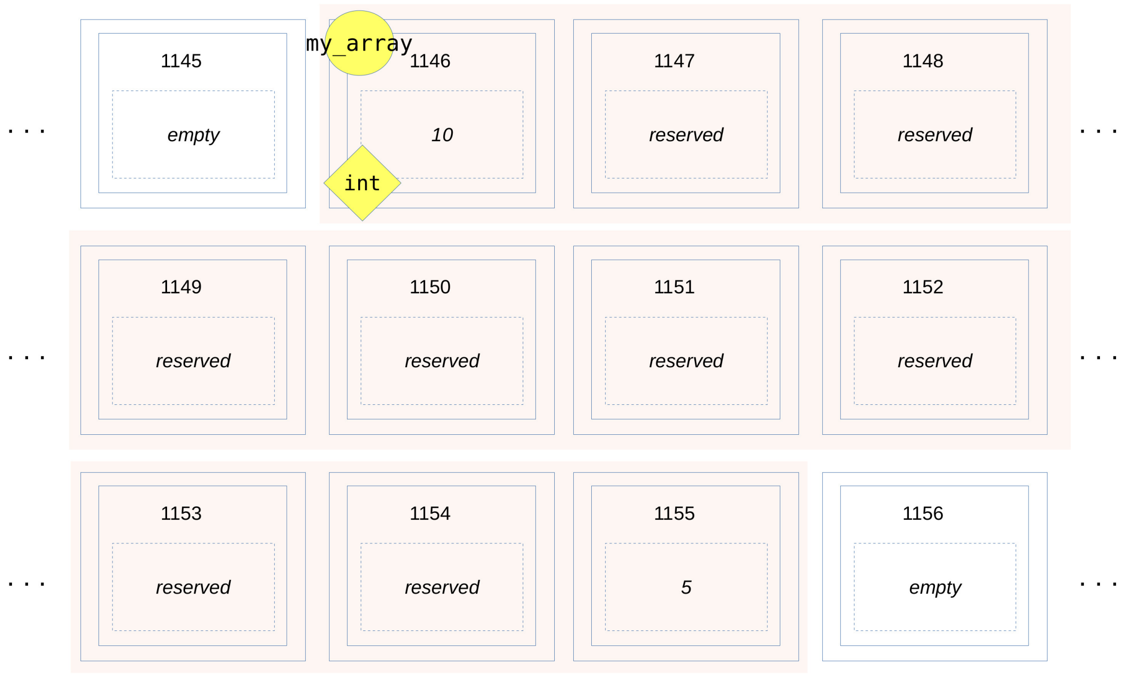


Figure 2.15: Memory model for an integer array with 10 entries, the first and last box were assigned values by the program, the rest remain **empty**.

All array declarations work the same way, the only thing that changes is the number of lockers reserved (which depends on the size specified in the array declaration), and the data type for the entries. You can create arrays with any valid data type supported by **C**.

Important facts about arrays that you must remember:

- The **size of the array is fixed**. Once you declare the array, you can not extend it. This is unlike the lists and dictionaries you are used to from Python which can grow whenever needed. You need to think carefully and in advance about how much space your program will need.
- The lockers reserved for the entries in an array **are always contiguous**, arrays can not have their entries scattered all over memory.
- Indexing is the same as in Python: **array[0]** is the first entry, **array[N-1]** is the last entry in an array with **N** entries. The only valid indexes are integers in **[0,N-1]**. Negative indexes are not allowed, neither are floating point values allowed as indexes.
- Array indexes are **offsets**, if you request to access **array[3]**, the compiler finds the first locker reserved for the array, and then looks for the box that is **3** lockers after it.
- All entries in the array have the same type, you can not mix data types within an array.
- **C** will not warn you or protect you from trying to access array entries outside valid index range. You must be extra careful with this, as it is a common source of bad bugs.

The last point in the list above is particularly important, as array indexing problems are among the most common, and often tricky to find sources of bugs in programs. Let's take for instance the sample array described above with **10** entries. We know this means that valid array indexes must be within **[0,9]**. But our program can request the following:

```
array[10]=5;     // Indexing beyond the array
```

This will in effect try to store an integer value of **5** in a location that is **10** lockers after the first box in the array. This box **has not been reserved for the array**, so our access request is **not valid**. At that point one of several things will happen (and we can not always predict which of them will occur, even for the same program, on the same computer, different things could happen at different times):

- The program will continue to run after the line that contains the invalid access. It will store a **5** at the locker you requested – **this is a bug**, we are overwriting something else. That box could be reserved for **another local variable or array**, it could be data from **another function in your code**, or it could be **empty space**. It will be hard to detect and fix this bug because the program continues as if there was no problem. The program will show unpredictable behaviour **and may or may not do the same thing at different times**.
- The code may **crash** (stop working before the normal end of its work) – this is obviously a bug, in this case the box we're trying to write to belongs to another program or is otherwise reserved. The computer's operating system terminates our program because it has tried to access data it doesn't own. Once more, this may happen unpredictably. **Sometimes the program may finish without crashing, other times it will crash**, or **it may work on one computer and not another**, or **it may work in Windows but not Mac or vice versa**.
- Another possibility is that the program continues past the line that has the invalid access, but the effect of changing a value in a box outside the array causes a problem later on, at a different point in the program. This can result in erroneous values being computed, unpredictable or unexpected behaviour from the program, or a **crash** at a different point in the code.

All of these are situations we must avoid. **Unpredictability** is not acceptable in a program, and array indexing issues are a common source of unpredictable behaviour. Therefore, be very, very careful when indexing into arrays in **C**.

Note

If your program is behaving in unexpected ways, check your array indexing and make sure you don't have indexes out of bounds at any point.

✍ **Exercise 2.3** Write a small program that creates an array for **10 floating point values**. Then fills this array with multiples of **pi**. That is:

```
        array[0]=1*pi;
        array[1]=2*pi;
        array[2]=3*pi;
        // etc... maybe you want to use a loop...
```

Have your program print out the entries in the array in separate lines once the array has been filled with the correct values.

✍ **Exercise 2.4** Modify the program you wrote for Exercise 2.3 so that there is a second array, this array should be of **integer type**, and the entries of this array are filled by **type-casting to int** the corresponding entry of the floating point array that contains the multiples of **pi**.

Make sure your program prints out both the **floating point** value, and the corresponding **int** value for each index in the arrays.

✍ **Exercise 2.5** Modify the program you wrote for Exercise 2.4 so that instead of using **type-casting**, the entries in the **integer array** are obtained by **rounding** the corresponding floating point value to the nearest integer.

You should write your own separate function that takes as input a floating point value, and returns the corresponding nearest integer.

2.4.1 Strings in C

One of the most common uses of arrays in **C** is for storing and manipulating **strings**. Indeed, in **C** a **string** is nothing more than an array of individual characters. This makes it very different from the strings you may have used in Python (and which provided you with all kinds of functionality). Strings in **C** are incredibly simple, and working with them becomes a matter of knowing how to access and manipulate entries in arrays.

Here's what a typical declaration for a **string** looks like in **C**

```
    char    a_string[1024];
```

Because **strings** are just arrays, they **follow exactly the same rules as arrays of integer, float, or any other data type**. Importantly, the size of the string is fixed and can not be changed after the string has been declared, and **initially the contents are junk**. The declaration above reserves space for up to **1024** characters, and we will not be able to store any strings longer than that (if we try, we will get into the same problems we described earlier regarding invalid indexing into arrays).

The string behaves exactly like any other array in **C**, so we can access and modify entries in the array as we normally would expect.

```
  a_string[0]='H';
  a_string[1]='e';
  a_string[2]='l';
  a_string[3]='l';
  a_string[4]='o';
  a_string[5]='\0';
```

The above code changes the contents of our string to contain the characters 'H','e','l','l','o','\0'.

Definition 2.2

The last entry is especially important, it is the **end-of-string delimiter***, in our program, we write it as a* **'\0'***. But it represents* **a single character***, not two. The* **'\0'** *is just a special sequence to tell the compiler we want to mark the end of a string.* ♣

Note

Why do strings need a delimiter?

Because we may want to store different text sequences within the same string array. The reason we declared the original array to have **1024** entries is that we want to have enough space that we can store a pretty large variety of text sequences without worrying they won't fit inside the string array.
Most of the text sequences we may be interested in **will be shorter than 1024 characters**. The example above contains only the text **Hello**, 6 characters altogether. But the string array contains **1024** entries and the remaining ones will be full of **junk**.
Any functions that need to work on the text of the string (for example, to print out what the string contains) need to know which entries contain **valid text, set by our program**, and **what entries are unassigned junk**.
The end-of-string delimiter is used to tell these functions where the valid text stops. Anything stored in the array after the end-of-string delimiter is ignored.

A common source of bugs stems from using the end-of-string delimiter incorrectly (e.g. putting in the wrong place, or alternately, forgetting to put it where it should be). **All valid C strings must have an end-of-string delimiter** just after the valid text portion of the string.

Question: If the string array contains **1024** entries, what is the maximum number of regular text characters that we can store in this string?

Of course, updating strings character by character would be a very annoying process. Luckily, there is a standard **C** library that contains functions you can use to manipulate strings.

```
#include<stdio.h>
#include<string.h>    // - This is the C library for string management

int main()
{
    char  a_string[1024];

    // Let's initialize a string - we can do that with the strcpy() (string copy) function
    strcpy(a_string, "This is a message for those learning C.\n");

    // Let's print out what we have stored in the string at this point:
    printf("%s\n",a_string);

    // Let's now concatenate another string to what we already have using the strcat() (string
        concatenate) function
    strcat(a_string, "Don't forget to practice using strings!\n");

    // Let's print out the string after concatenation
    printf("%s\n",a_string);

    return 0;
}
```

Compiling and running the program above produces

```
>./a.out
This is a message for those learning C.

This is a message for those learning C.
Don't forget to practice using strings!
```

A couple of things worth noticing:

A single string can contain multiple lines of text (we can split text into separate lines by using the special end-of-line character **n**).

The functions in the **string library** automatically make sure the end-of-string delimiter is properly placed after each manipulation of the string.

In the examples above we used **constant string values** (the text sequences within double quotes in the program above) to **initialize** the string and to **concatenate** onto the string. But functions in the **string library** can use other **string arrays** as well, for instance, we can copy one string onto another with **strcpy()**, and we can concatenate the contents of two string arrays using **strcat()**.

There are many more functions in the **string library** that you will find useful. For example, **strlen()** returns the **length of the string** - this is the number of characters of valid text, not including the end-of-string delimiter. Another useful function **strcmp()** can compare the contents of two strings to tell you whether they are identical or different.

If you need to look up how to use these function, a handy resource is here: https://www.cs.bu.edu/teaching/cpp/string/cstring/. This is from the computer science department at Boston University. There are many other references and examples for using string library functions, so as ever, if you have a problem, Google is your friend!

Note

A final thought regarding strings: **it is easy to run intro trouble with index-out-of-bounds issues** when manipulating strings. In particular, if we are concatenating strings, it is easy to end up with a text sequence that is too long for the array that is meant to store it. This would create mayhem in memory as the string manipulating functions will try to write to parts of memory that are not part of the string. You **must be careful** to ensure all the text sequences your program will work with fit within the arrays reserved for the strings themselves.

2.5 What is a pointer, and why do we need them?

Let's review the original program we wrote to compute the square of a number:

```
#include<stdio.h>

int square(int y)
{
    return y*y;
}

int main()
{
    int x;
    x=9;
    x=square(x);

    return 0;
}
```

You have seen this program before, **main()** will reserve space in memory for one integer variable called **x**, and set it to **9**. When we call **square()**, it will reserve space for an integer parameter called **y** and for an integer **return value** which will be set to **y*y**. This is shown in Fig. 2.16.

The point we need to make here is that the only way for **square()** to change what is stored in **main()'s local variable** *x* is by returning a value. This is the consequence of variables being local to functions, as we have discussed in detail previously. This is perfectly fine for a wide range of functions we will ever need to implement, and it is a **safe** way to write code, because the locality of variables ensures functions can not go around changing data that they do not own.

However, there are situations where we want a function to be able to **directly access, read, or modify** information that is stored somewhere else, and that is owned by a different part of the program. For example, this is a common situation when working with **arrays**. Let's say you have a program in which **main()** declared and is using an array with one million floating point values (for example, this could be a simulation of weather patterns, or it could contain statistics about stock prices, or it could represent the location of vehicles in a city grid). Now imagine that every time **main()** calls another function to do some work that requires the array **we make copies of the array** so each function has its own **local copy** as dictated by the principles of **local variables** and **passing input arguments by value**.

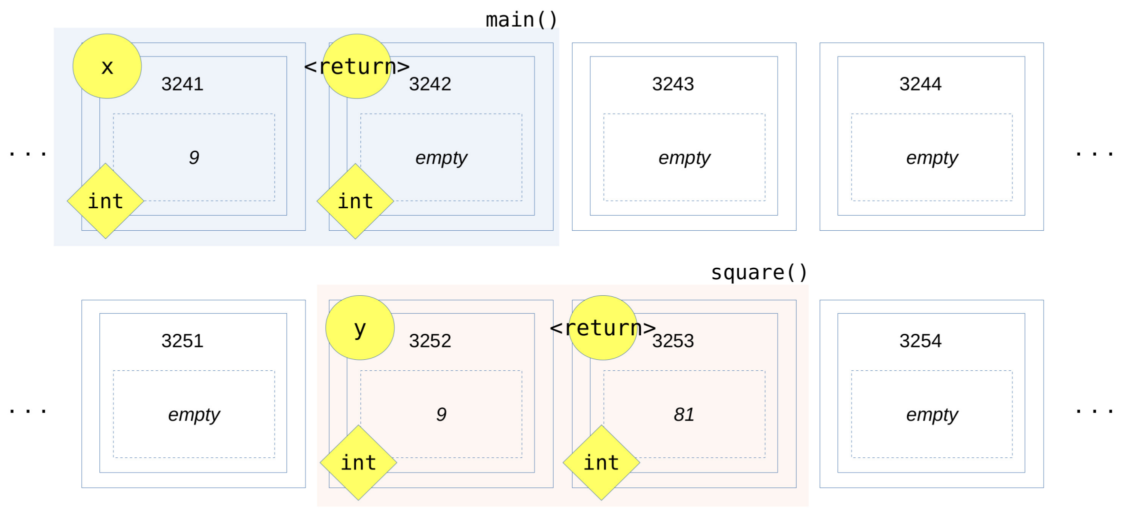


Figure 2.16: Memory model for the short program that computes the square of a number. This is what memory would look like **just before the memory used by** *square()* **is released**.

Clearly, doing this would result in a lot of space being used for copies of the array, and a lot of time and work would be spent simply creating and maintaining these copies so that they are **in sync** - all of these copies represent the same data, so they should contain the same values once work on them has been done.

Note

This is not a good way to handle large collections of data. An array with one million entries is actually fairly small, by today's standards. A single video file with about 30 minutes worth of video can easily require one thousand million array entries to fully load into memory.

In **C**, there is a **simple and intuitive way** to allow a function to access information it doesn't own, it's called **a pointer**.

To understand what a **pointer** is, let's start with an analogy to our locker room model for memory:

You reserve a locker at your gym to keep your clothes and toiletries. You know the locker number so you know where your locker is, and the gym **reserved the locker for you** *by giving you a key so only you can open the locker and access what's inside. As you are exercising (maybe playing basketball) with your friends, you realize you forgot your water bottle in the locker. Lucky for you, one of your friends is heading there to fetch their phone* **so you ask them to go to your locker and get your water bottle for you**. *You* **tell your friend your locker's number** *so they can find it -* **in effect you just gave your friend a pointer to your locker!**, *and you share your key so they can open the locker and get the bottle for you. One should always stay well hydrated.*

Definition 2.3

In **C**, *a pointer is simply a* **variable** *that contains a* **locker number** *that* **indicates where some information the program needs to use is stored**. *Like all other variables declared by a function, it* **will have its own box in the memory model**, *and has an* **associated data type** *that tells you* **the data type of the information stored in the locker** *that the pointer is referring to.* ♣

There is nothing mysterious about **pointers**, they are nothing more than **locker numbers** we use to go find a specific box whose data we need.

In **C**, pointers have their own particular (and, to be fair, somewhat clunky) syntax. There are three fundamental operations we need to learn to do with pointers: **declare a pointer**, **assign a locker number to the pointer**, and **use the pointer to access information**. Let's see how these work by way of an example:

Example 2.6

```
#include<stdio.h>

int main()
{
    int my_int;
    int *p=NULL;            // A pointer to an int!

    my_int=10;              // Change the value in 'my_int' directly
    printf("My int is: %d\n",my_int);

    p=&my_int;              // Put 'the address of my_int' in pointer p
    *(p)=21;                // Use the pointer to change the value of 'my_int'
    printf("My int is: %d\n",my_int);

    return 0;
}
```

Let's see what's happening here. First, **main()** reserves space for two variables, an integer variable called **my_int**, and a **pointer to an integer variable** called **p**.

The **'*'** in the line:

```
int   *p=NULL;
```

specifies that we are declaring a **pointer variable**, and the associated data type tells us **the pointer will keep track of the locker where we stored an int**. Importantly, the pointer is initialized to **NULL** to indicate it is initially **not assigned** to anything (we have not put a valid locker number in it just yet). In memory, this looks as shown in Fig. 2.17.

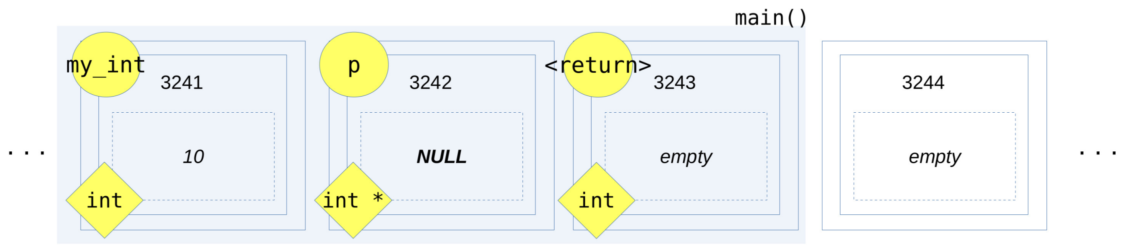


Figure 2.17: Memory model for the program in Example 2.6 up to the line **my_int=10;**.

The next line is something we haven't seen before:

```
p=&my_int;        // Put 'the address of my_int' in pointer p
```

this should be read as **get the address of the variable called** *my_int* **and store it in** *p*. Whenever you find a statement like this in **C** you should **translate it into the corresponding sentence in English** so that you clearly understand what the program is doing. The **'&'** symbol is read as **'the address of'**. The effect of this line of code is that the compiler **looks up the locker number for the box where** *my_int* **is stored** and copies that locker number into **p**. This is exactly the same thing that happens when you give your friend the number of your locker at the gym. The result of this is shown in Fig 2.18.

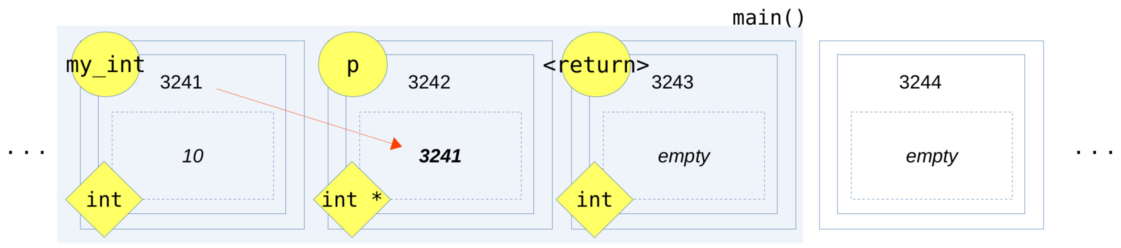


Figure 2.18: Memory model for the program in Example 2.6 up to the line **p=&my_int;**.

So now, our pointer **p** stores the value of the locker number reserved for **my_int**, and **we can use the pointer to access or even change what is stored in that locker**.

The next line does exactly that:

```
*(p)=21;
```

This should be read as **go to the locker number specified by (p) and store 21 in there**. Whenever you see a sentence like this:

```
*(a pointer or pointer expression) = expression
```

or

```
variable = *(a pointer or pointer expression)
```

You should read the **'*'** as **the contents of** (whatever is in the pointer expression within the parentheses).

So the instruction ***(p)=21;** becomes **make the contents of (p) equal to the value 21**. We know that **p** has the value **3241**, so the instruction really says **make the contents of box (3241) equal to the value 21**. We are telling the compiler to go to a specific locker, and put a particular value in that locker. This is shown in Fig. 2.19. Thereafter, if we print the value of **my_int**, we will see it is **21**.

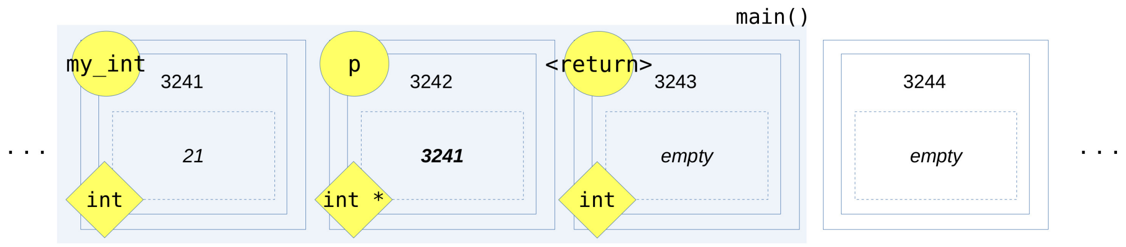


Figure 2.19: This is the result of using pointer **p** to change the value of **my_int**.

What we just did is: **we used a locker number to directly access a specific box in memory**. We did that without using the name of the corresponding variable which is important because we said before that each box in memory can only have one tag. However, now we have two different ways to access box **3241**. One is by using the local variable **my_int**, and the second one is by using pointer **p**. There really is **no practical difference** in terms of using one or the other of these two methods, both will allow you to read or modify the value stored at box **3241**. However, the advantage of using a pointer is that we can have the same pointer variable **point to different lockers** at different points in the program. So, unlike the local variable **my_int** which is always attached to the same box, our pointer **p** can be used to access different data items at different points in the program.

Note

Using a **pointer** to access a locker directly will work **as long as the locker is reserved by some function in our program**. We can not go poking around memory that is **not reserved by our program** by using pointers to lockers the program doesn't own. If we try that, the computer's operating system will terminate (end/crash) our program. This is a common bug. **If your program does not complete correctly** (it ends without finishing its work, seemingly for no reason) or if the computer reports a **segmentation fault**, that is a clear sign that there is a problem with the way your program is accessing memory locations. This can be because the program is using pointers to lockers that it shouldn't access, or it can happen if the program is using invalid indexes into arrays. So, if you experience these problems, the first thing to do is to carefully check code that uses pointers or arrays and make sure it is not accessing memory locations that are not valid and correct for your program.

Note

As mentioned above, the syntax for **pointers** in **C** is a bit clunky. In particular, you have to be careful with the meaning of the * symbol. It is used in two very different ways in the context of pointers:

```
float  *fp;       // This tells the compiler you want to declare a pointer called 'fp'
                  // which will point to a floating point data item

*(fp)=3.1415926;  // This means 'assign to the contents of (fp) the value 3.1415926' here
                  // the '*' is used to access data in a particular locker
```

✍ **Exercise 2.6** In a memory model diagram, show what's happening in memory with the following small program that uses pointers. Indicate what the final value for **x** and **y** are.

```
#include<stdio.h>
int main()
{
    int   x,y;
    int   *p=NULL;

    x=5;
    p=&x;
    *(p)=3;
    y=*(p);
}
```

Note

One more note regarding good programming style when declaring pointers. You should use

```
    int *p;
```

instead of

```
    int* p;
```

both of which compile, and both of which you can find in textbooks and programming manuals. The reason for this is using **int*** (or any other data type with a * attached to it) is ambiguous. Here's why:

```
    int a,b;
```

tells anyone reading the code that you are declaring two integer variables, **a** and **b**. So, a declaration that reads

```
    int* p,q;
```

would reasonably be taken to mean you are declaring two integer pointers **p** and **q**. However, **this is not correct**. The preceding line will declare a single integer pointer **p**, and a regular integer variable **q**. This can easily be a source of misunderstanding. Conversely, any of the following declarations are completely unambiguous:

```
    int *p, *q;
    int *p, q;
    int p, *q;
    int p,q
```

in each case, it is perfectly clear which variable is a pointer and which is just an int. Make sure to write **code that is clean, easy to read, and contains no ambiguity about what you mean to do**.

2.5.1 Passing pointers to functions

As we mentioned before, a function can not directly use, access, or modify data that is external to itself. It can only receive data via its **input arguments**, and can only send data out to the rest of the program via its **return value**. This is the correct way for the function to work in many situations, and it is also good programming practice because it prevents the function from accidentally changing data external to itself in unexpected ways. However, we will find situations in which we need a function to directly access and/or modify data that is external to the function.

Let's update our example with the **square()** function to understand how this may work, and once we see the process in action, we will study the most common situation that requires us to allow functions to access information owned by a different part of the program. Consider the version of the **square()** function shown in the example below:

Example 2.7

```
        #include<stdio.h>

        void square(int *p)
        {
```

```
        int x;         // This is unnecessarily long,
        x=*(p);        // but we want you to see every
        x=x*x;         // step of the process.
        *(p)=x;
    }

    int main()
    {
        int my_int;
        my_int=7;

        square(&my_int);       // This reads 'take the address of my_int
                               // and pass it to square()'
                               // It's like telling your friend your locker
                               // number at the gym.

        printf("The final value for my_int is: %d\n",my_int);

        return 0;
    }
```

In the memory model, the first thing that happens is **main()** reserves space for a local variable **my_int**, and for its **return value**. Then it sets **my_int** to 7, the result is shown in Fig. 2.20.

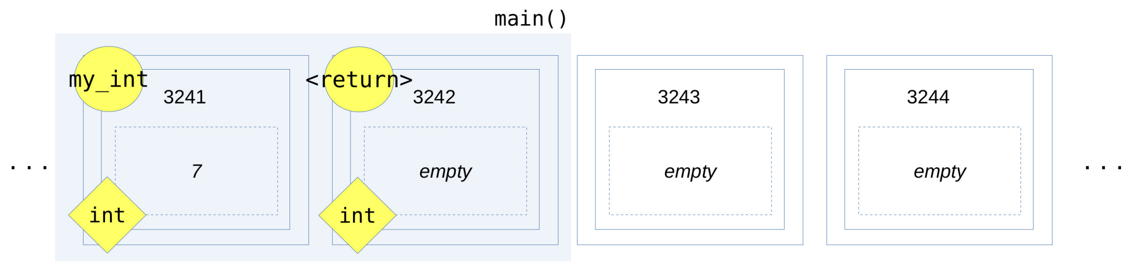


Figure 2.20: Memory model for the code in Example 2.7 after the line **my_int=7;** is processed.

The next line in the code: **square(&my_int);** is the call to the square function, it does the following:

- Reserve space for **square()**. One box for the **input argument p** which is a **pointer to int**, then a box for the local integer variable **x**. Now we notice something we haven't seen before. The function declaration states the **return value** is of type **void**. This effectively means **the function does not have a return value**. That means **there will be no box reserved for the return value of** *square()*.
- Pass the required input argument to **square()**. The function is being called with **(&my_int)** as argument. This means **get the address of** *my_int* **and pass it to** *square()*. The **input argument** *p* is a pointer, so we need to provide it with **an address, a locker number**.
- The program then continues with the code in **square()**.

The situation in the memory model will be as shown in Fig. 2.21. Pay close attention to the value stored in the box labeled **p**, it now contains **the locker number** for **main()'s** local variable **my_int**.

The next couple lines in **square()** do the actual work, let's look at them one by one:

```
    x=*(p);
```

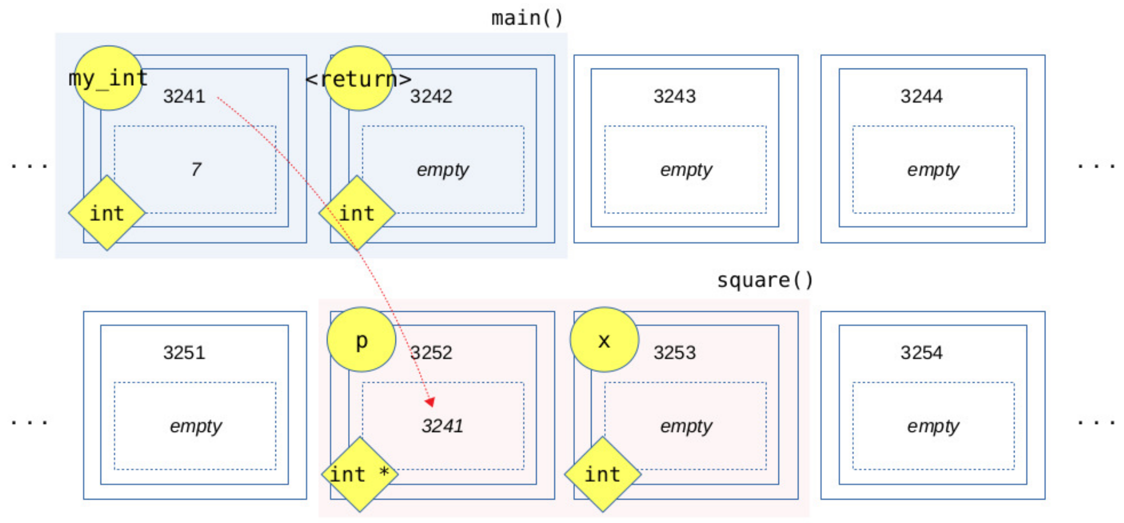


Figure 2.21: Memory model for the code in Example 2.7 once the function **square()** is called. Notice **there is no box for a return value for** *square()*.

this reads **get the contents of (p) and store that value in** *x*. Since **p** now has the locker number for **my_int**, this line effectively becomes **get the contents of locker (3241) and store that value in** *x*. This stores a **7** in variable **x**. The next line **x=x*x;** simply computes the square and makes the value of **x** equal to **49**. Finally, the last line

```
   *(p)=x;
```

updates **my_int**. The line reads **make the contents of (p) equal to the value in** *x*, or, if we think of **p** as a locker number, **make the contents of box (3241) equal to the value of** *x*. The end result will be that **main()'s** local variable **my_int** will be updated to the value **49**. The situation in the memory model just before the memory reserved for **square()** is released is as shown in Fig. 2.22.

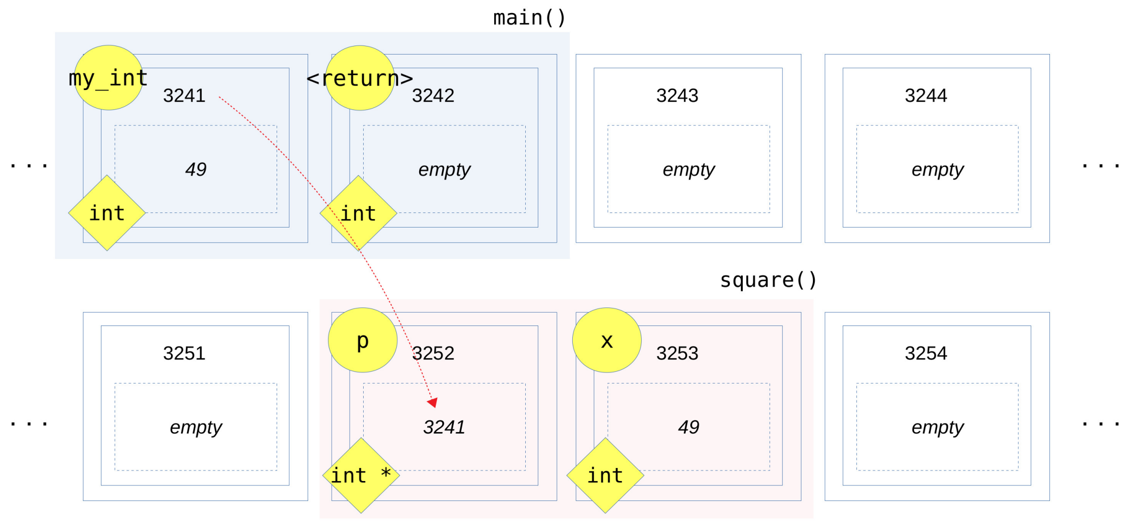


Figure 2.22: Memory model for the code in Example 2.7 after function **square()** has been processed, **right before the memory reserved for square() is released**.

Note

A small note regarding the code above. We could have written a much shorter, but also harder to understand version of **square()**. The short version doesn't use the **local variable x**. Instead, it contains a single line of code:

```
        *(p)=(*(p))*(*(p));
```

Which should be read as **make the contents of (p) equal to the value of the contents of (p) times the contents of (p)**. This is correct, and shorter, but also hard to parse because the syntax is clunky. Notice the use of **parentheses** to clarify what is being multiplied. In particular, the parentheses show that **(*(p))** is one thing being multiplied with another **(*(p))** thing.

You could avoid the parentheses altogether and write:

```
        *p=*p**p;
```

which will **compile and run** just fine, but is **very hard to read and understand** for anyone other than whoever wrote this **pretty ugly code**. So the message is this: Please write code using pointers **as clearly and unambiguously as possible**. The syntax is clunky, but you can use **parentheses** to help group things into **units that make sense** and can be more easily read and understood by a human going through the code.

From the example above, it's important to remember that **passing a pointer into a function** gives the function access to information that would otherwise be out of reach. This is both a powerful feature, and a possible source of all kinds of bugs. So **use this with care**. As noted earlier, most functions that perform simple computations and can return the result **should use a return value and not have access to external data**. However, handling larger amounts of information such as may be stored in arrays or strings is best done by using **pointers**. Let's now see how that works in **C**.

2.5.2 Pointers and Arrays

At the start of this section we briefly discussed that with arrays (or strings) we want to avoid having to make copies in order to provide a function with the information needed to do some work. Copying arrays would result in an often unnecessary waste of space as well as time. So, in **C**, by **design**, arrays will be passed onto the functions that use them as **pointers**.

Definition 2.4

In **C**, *arrays are* **passed by reference**. *This means that whenever we pass an array (or string, which is just an array of* **char**) *to a function, the function receives* **a pointer to the first entry in the array**. *The array* **is never copied**. *The function thereafter can* **directly access/modify** *the contents of the array that was passed to it, regardless of which part of the program declared and owns the array.* ♣

There are a couple of important ideas to keep firmly in mind regarding arrays passed on to functions:

- The function will only get **an address/locker number**. None of the values of the array are ever copied or passed on to the function.

- The function **must also receive** the **size of the array**. This is because arrays in **C** are only collections of boxes. They do not contain information about their size, this has to be provided separately so the function knows how many items of data it has to work with.
- The function will access the array normally, that is, it doesn't have to use **pointer notation** to access information in the array. This is done for the convenience of the programmer. What actually happens is that **the compiler translates all array accesses to the equivalent pointer expressions** without us needing to worry about it. This is also the case in the original function that declared and owns the array.

Let's have a look at an example of how this works:

Example 2.8

```
#include<stdio.h>

void square_array(int array[], int size)
{   // Square the value of each entry in the array
    int j;

    for (j=0; j<size; j=j+1)
    {
        array[j]=array[j]*array[j];
    }
}

int main()
{
    int i;
    int my_array[5];

    for (i=0; i<5; i=i+1)
    {
        my_array[i]=i;
    }

    square_array(my_array,5);

    for (i=0; i<5; i=i+1)
    {
        printf("%d squared equals %d\n",i,my_array[i]);
    }

    return 0;
}
```

Compiling and running the code above produces:

```
./a.out
0 squared equals 0
1 squared equals 1
2 squared equals 4
3 squared equals 9
4 squared equals 16
```

Let's see what's happening in memory when we execute this code. First, **main()** reserves space for a 5-entry **integer array** and fills it up with values from **0** to **4** (remember, array indexes for an array of length **k** go from **0** to **k-1**). This is shown in Fig. 2.23.

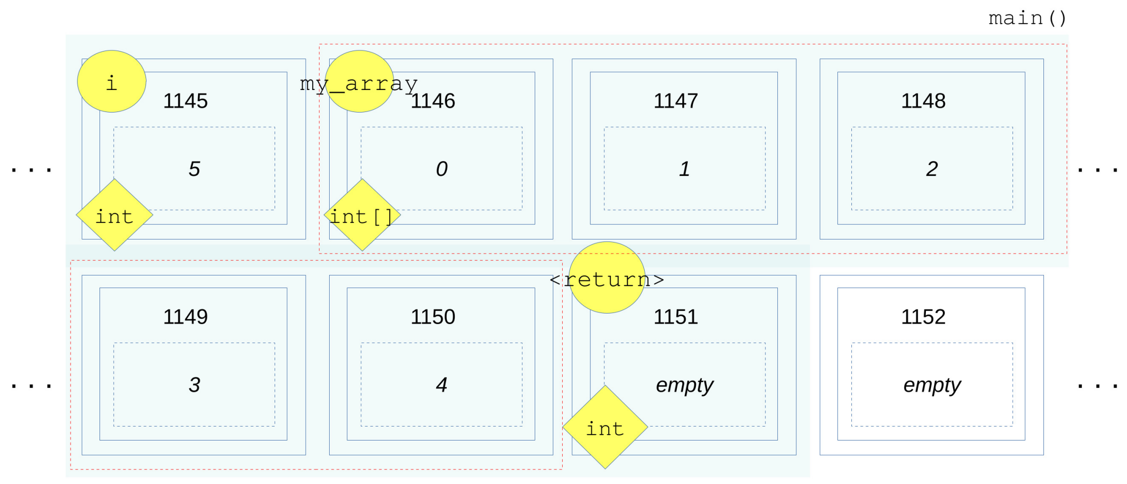


Figure 2.23: Memory model for the code in Example 2.8 after **main()** has declared and filled a small integer array.

The memory reserved for the array is marked by the red-dashed lines (as expected, it's contiguous, and only the first box is tagged with the name and type for the array). When the function **square_array()** is called, the following happens:

- The compiler reserves space for the input parameters. The first one is declared as an **int array** but notice **no size is given** because the function doesn't know what the size of the array was when it was declared, and also because we would want this function to work with arrays of any size. Because **arrays are passed by reference** the compiler will reserve **a single box** in which it will place **the address of the first entry of the array**.
- The compiler then reserves space for the integer input parameter **size**, which will provide the size of the array the function will be working on.
- Then the compiler reserves space for local variable **j**.
- Finally the program continues with the instructions in **square_array()**.

In the memory model this will have the effect shown in Fig. 2.24.

Things to note

- Function bf square_array() only reserved 3 boxes in memory, despite the fact that it will be working with a 5-entry array. The function would reserve only these 3 boxes even if the input array had millions of entries
- The box tagged **array** has a type **int[]** to indicate it is an array, however, what it actually contains **is the address (or locker number) of the original array in** *main()*

The last point above deserves comment. The box for **array** looks like an actual array would, and the code in the function **square_array** uses the **array** variable as you would use any other array. This **is done for the convenience of the programmer**. In reality what the **array** box contains is just a **pointer to the first entry in the array from** *main()*.

After the compiler has reserved space for **square_array**, the function goes about its work. The important thing to understand here is that the line

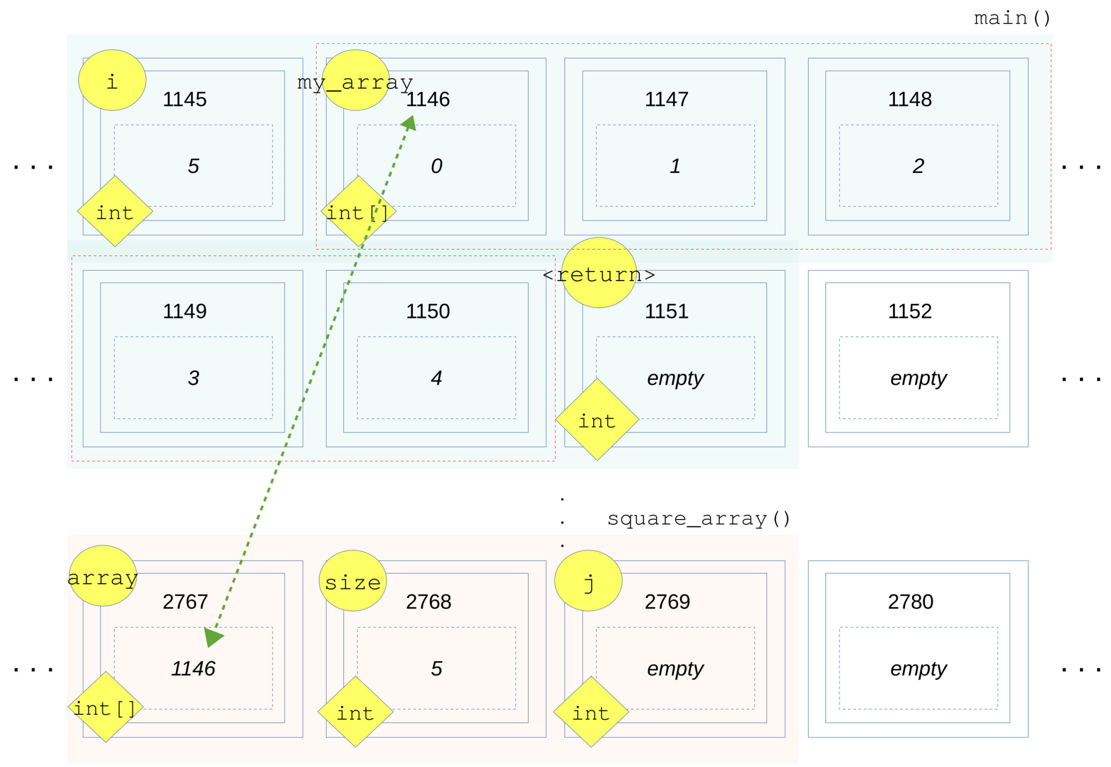


Figure 2.24: Memory model for the code in Example 2.8 after the call to **square_array()** has reserved all the space the function will need.

```
    array[j]=array[j]*array[j];
```

is **directly working on the data stored in** *my_array* **which was declared and is owned by** *main()*. Passing the array to the function is exactly the same thing as **giving the function a pointer** so it can work directly with the data in the original array.

Question: What memory location is being accessed if **square_array()** stores something in **array[3]**?

Question: What do you think will happen if **square_array()** attempts to store a value in **array[11]**?

After the function has completed its work, we expect the contents of memory to look as shown in Fig. 2.25.

The example above shows how useful pointers can be - without them working with arrays would be much more involved. The same is true for all work we will do using strings. Indeed, if you look at the definitions for functions in the string library, they all take as input parameters pointers to char arrays.

2.5.3 Using pointers and offsets to access array data

As noted above, when we pass an array into a function what the compiler does is provide a **pointer to the first entry in the array** so that the function can directly access array data. The **C** compiler then translates all the array instructions in the function code into the corresponding expressions using **pointers**. We can choose to write code

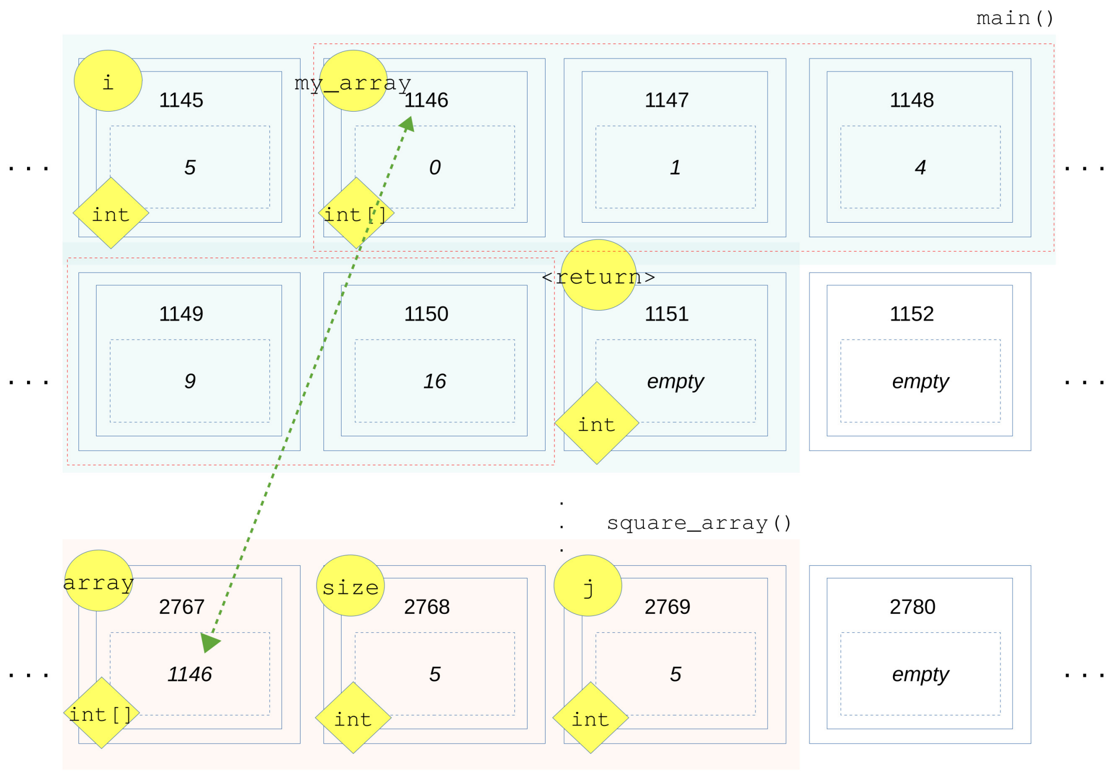


Figure 2.25: Memory model for the code in Example 2.8 after the call to **square_array()** has completed but **just before the memory reserved for** *square_array()* **is released**.

that uses **only pointers** instead of **array expressions**, and this is the more common way to work with arrays and strings in **C**. Let's see how that works, first by using pointers with a small example array, and then by rewriting the **square_array()** function in Example 2.8 to use pointer expressions instead of arrays.

Example 2.9 The program below uses pointers and offsets to change entries in an array without using array notation

```
#include<stdio.h>

int main()
{
    int  array[10];
    int  *p=NULL;

    p=&array[0];     // Get address of the first entry in the array

    *(p+0)=5;        // Set the contents of array[0] to 5
    *(p+4)=10;       // Set the contents of array[4] to 10fs

    printf("array[0] is %d, array[4] is %d\n",array[0],array[4]);

    return 0;
}
```

The example above initializes a pointer to the address of the first entry in **array** (the line **p=&array[0];** is read as **take the address of** *array[0]* **and store it in** *p*. Given a pointer into an array, we can **use an offset** to access any of the entries in the array. Offsets are exactly equivalent to **array indexes**, so **(p+4)** is referring to the data item stored four boxes **after the first entry in the array**. The expression

```
*(p+4)=10;
```

is read as **make the contents of** *(p+4)* **equal to the value** *10*. Since **p** contains a locker number, **(p+4)** gives the location of a box four locations after the first entry in the array.

The instruction

```
*(p+7)=2;
```

is exactly equivalent to

```
array[7]=2;
```

in fact, as we have discussed above in the context of passing arrays to functions, the compiler will convert the expression that uses array notation to the corresponding expression that uses pointers when translating your program to **machine language**.

It's important to develop the ability to use pointers and offsets to access information, and we will have plenty of opportunity to practice this skill through the book. For now, please keep in mind the following important points that apply to the use of pointers and offsets to work with data in arrays:

- Similarly to array indexes, the compiler will not tell you if a **(pointer+offset)** expression results in your program trying to access a box it doesn't own, or a box that is not part of the array you meant to work with. You can easily get in trouble by using the wrong offset together with a pointer, **so be very careful** that your offsets and pointers are always correct and the resulting access is valid.
- Problems with accessing data via pointers and offsets can easily result in **unpredictable behaviour**, or your program may be **terminated by the computer** (e.g. you get a **segmentation fault**, or the program simply stops and nothing else happens, not even an error message).
- Negative offsets can be used, though they do not make sense if your pointer contains the address of the first entry in an array. We will come across situations where negative offsets have a role to play in our program, just be careful in the meantime.

Let's now do one more example and see how we would rewrite the **square_array()** function from Example 2.8 so that it uses pointers and offsets, instead of array notation.

Example 2.10

```
#include<stdio.h>

void square_array(int *array, int size)
{   // Square the value of each entry in the array
    int j;

    for (j=0; j<size; j=j+1)
    {
        *(array+j)=(*(array+j)) * (*(array+j));
    }
}

int main()
{
    int my_array[5];
    int i;

    for (i=0; i<5; i=i+1)
    {
        my_array[i]=i;
    }

    square_array(&my_array[0],5);

    for (i=0; i<5; i=i+1)
    {
        printf("%d squared equals %d\n",i,my_array[i]);
    }

    return 0;
}
```

The code in Exercise 2.10 is very similar to the code in Exercise 2.8, but the differences are important and worth spending a moment on Firstly, the function declaration:

```
void square_array(int array[], int size)    // Using array notation

// becomes

void square_array(int *array, int size)     // Using pointers
```

The change is mostly a matter of syntax. We know that arrays are passed by reference, and that passing an array (first version of the function declaration) results in a single box being reserved, and in that box the compiler puts the address of the first entry of the array. The second version of the function declaration makes this explicit: The function expects to receive the address of an integer data item. Both functions will receive exactly the same thing, but the pointer version of the function makes explicit the fact that what the function is getting is just a locker number.

The second change is within the function itself:

```
array[j]=array[j] * array[j];           // Using array notation

// becomes

*(array+j)=(*(array+j)) * (*(array+j)); // Using pointer notation
```

which is exactly the same thing - the first version of the code (using array notation) will be converted by the compiler to an expression that looks like the second version, using pointers and offsets. The only difference is what the programmer would read, the array notation is perhaps easier to understand, but both do exactly the same thing.

Finally, there is a small change in the way the function is called:

```
square_array(my_array,5);        // Passing an array

// becomes

square_array(&my_array[0],5);    // Passing the address of
                                 // The first entry in the array
```

The first version (with array notation) takes advantage of the fact that the compiler knows to pass arrays **by reference**, so when we call a function using the array name, it provides the function with the address of the first entry in the array. The second version (with pointers) explicitly passed **the address of my_array[0]** - which is in effect the same thing but we are making explicit the fact that we are passing down a locker number. Because the compiler knows to pass arrays by reference, we could call the pointer version of **square_array()** exactly as we would call the version that uses arrays: **square_array(my_array,5);** works with both versions.

The notation can be a bit confusing, so whenever you're working with an array, remember the following:

```
my_array;      // The name of the array, same as the address of the first entry in the array
&my_array[0];  // 'The address of' my_array[0], also the first entry in the array
my_array[0];   // The value stored in the first entry of the array, *this is NOT an address!*
```

2.5.4 Summary

What are pointers? They are variables that hold the memory address (locker number!) of the contents of a variable, array, string, or data structure we want to share between functions in the program.

Why are they useful? Because they allow functions to access and modify data that has been declared outside of the function's code. Because they allow us to share information between functions without making extra copies of the data we are sharing.

What is the syntax for declaring a pointer variable?

```
  type   *pointer_name=NULL;
```

for example,

```
  int  *p=NULL;         // A pointer to an int
  float *fp=NULL;       // A pointer to a float
```

What is the syntax for assigning a pointer to a variable?

```
  pointer = &variable;
```

Which is read as **assign to** *pointer* **the address of** *variable*, for example

```
  float pi;
  float *fp=NULL;

  pi=3.1415926535;
  fp=&pi;
```

initializes a float called **pi**, and sets the pointer **fp** to the address of **pi**.

How do we access or update boxes in memory using pointers?

```
  *(pointer+offset)=expression;  or  variable=*(pointer+offset);
```

the expression on the left is used to store a value at the locker specified by **(pointer+offset)**, the expression on the right is used to get the value stored at locker **(pointer+offset)** so we can assign it to a variable. Examples:

```
float pi;
float *fp=NULL;

fp=&pi;
*(fp)=3.1415926535;
```

The above uses the pointer to update the value of **pi**.

How do we obtain a pointer to an array? Get the address of the first entry in the array. Like so:

```
  int array[10];
  int *p=NULL;
  p=&array[0];
```

or, equivalently

```
  int array[10];
  int *p=NULL;
  p=array;
```

the above works because the compiler takes the array name to be equivalent to a pointer to the first entry in the array.

How do we update entries in an array using a pointer? Since the pointer has the location of the first entry in the array, we can use the pointer plus an offset to access and/or update any entry in the array like this:

```
int array[10];
int *p=NULL;

p=&array[0];

// Set the first entry in the array to 7
*(p)=7;

// Set the second entry in the array to 10
*(p+1)=10;

// Set the third entry in the array to 15
*(p+2)=15

// Set the last entry in the array to -15
*(p+9)=-15;

// The next two lines are an example of invalid
// (pointer+offset) combinations - they are a
// bug!
*(p+15)=25;
*(p-2)=0;
```

as noted earlier, we have to be careful that our **(pointer+offset)** expressions do not result in access to a box that is not valid. The last two lines would clearly attempt to store data in lockers not reserved for our program, so any of those lines would be expected to cause a problem.

Final notes on pointers:

We will be using pointers for the better part of the course. You should make yourself comfortable with them, practice using them for passing parameters to functions, for having functions manipulate and change data defined outside of them, and to access and change the contents of arrays, both by passing them to functions that manipulate them, and by using pointers and offsets to access data in the arrays.

You should always make sure that

- Pointers are initialized to **NULL** when you declare them - this will save you no end of trouble as you can check whether the pointer has actually been assigned to something or not.
- Every function that receives a pointer as a parameter must check that the input pointer is not **NULL**. You can not access a **NULL** pointer and trying to do so will crash your program.
- When using pointers and offsets to access data, you must ensure the offsets used are within bounds for the array you're indexing into.

✍ **Exercise 2.7** Let's put everything you've learned in this section together. Write a program that:

- Declares a string in **main()**, you can fill that string with any text you wish, for example: **"Now I know how to program in C!"**.
- Write a function that **takes the string as input**, and reverses the string. Mind the fact that the length of the string may be different than the size of the array, and that the reversing process should not move the **end-of-string delimiter**.
- Try to implement the reverse function using **(pointer+offset) notation** only.
- Have your code print the reversed string.
- Write a function that takes takes as input **the string**, a **query character**, and a **target character**, then goes over the string and replaces any occurrence of the query character with the target character.
- Try to implement the replace function using **(pointer+offset) notation** only.
- Have your code print the string after replacing all occurrences of **a** with **i**.

For instance,

```
        If the input string is "Hawdy! that strange pointer was NULL"

        then the reverse function would print:

        LLUN saw retniop egnarts taht !ydwaH

        and after replacing ’a’ with ’i’ the program would print

        LLUN siw retniop egnirts tiht !tdwiH
```

That's it! Have a go at it. You may want to read the last part of this Chapter on building code that works, and apply the advice there to solving this exercise.

And finally:

You've worked very hard in this Chapter to become familiar with C and how it works, so you deserve a moment to celebrate.

Go have a nice lunch, watch a movie, play sports, spend time with your friends, or something equally fun.

Congratulations! Now you know how to program in C!

Figure 2.26: Climbing a mountain is hard work, but the view from the top is worth it! *Photo: WolfmanSF, Wikipedia Commons, CC-SA 3.0.*

2.6 Model versus Reality

Through this Chapter, we developed a memory model to help us understand how information is being manipulated by our program, how variables, arrays, function arguments, return values, and pointers work. But,you should keep very clearly in your mind the following idea: The **memory model** we are studying here is exactly that - **just a model**. It is **an abstraction** intended to help you understand how information is organized and manipulated by a **C** program as it goes about its work. It is a good model in that it captures correctly the way in which memory is organized inside your computer, but by necessity, **it simplifies or removes some of the details** because they are neither relevant at this point, nor helpful in understanding how a C program works. However, it's important to make sure we understand what parts of the model are entirely accurate, and which are simplified.

Our **memory model** accurately describes:

- That memory is organized as a large collection of **sequentially numbered boxes** that can be used to contain data.
- That **each variable** our program declares **is assigned its own box**, its box can not be shared by any other variables, and it's tagged with the variable's name and data type.
- That **variables declared by the same function** in our program **will be assigned neighbouring boxes** in the memory model.
- That the **return value** of a function **will have its own reserved box**.
- That **variables and input arguments are local** to a function. Only the function that declared them knows their name, type, and can access or store values in their corresponding boxes by using the variable's name.
- That **arrays** are simply collections of **consecutive boxes** with the same **data type**.
- That **pointers are just variables** so they have **their own box** in memory, that box will store **the locker number** of another variable or array we want to work with.

- That memory reserved for a function **is released** after the function ends.

Some of the details that the **memory model** simplifies (removes or abstracts away):

- In our model neighbouring boxes have locker numbers that **increase by 1**, and **we assume boxes can hold any data type**. In reality each data type has a different size and the compiler needs to reserve the correct amount of space. This means that if we look at the locker number for two neighbouring variables **A** and **B** inside the computer, their locker numbers may be different by more than 1 unit. This is not relevant to us as it is handled automatically by the compiler.
- The order in which variables are placed in memory. As noted above, variables declared by a function will be placed in contiguous locations, but the order is up to the compiler and may be different from the order in which we declared them in our program. Assuming that the variables will be placed in the order in which they were declared is **reasonable and often accurate**, but keep in mind the compiler may choose to reorder things if it yields better code in **machine language**.
- That the **return value** is placed at the end, after all the input arguments and variables for the function. Once again this is often correct, but once again the compiler may choose to change things around if it yields better code after translating to **machine language**.

If you are interested, you can learn in detail how the parts of the memory model that were simplified actually work. This is part of what any good course in Computer Organization would cover in detail, so if you are curious, find a good book on that topic and read on. For the purpose of this book, we will work with the memory model as described above, and consistently follow the rules we described for how variables are ordered in memory.

2.7 Additional Exercises

Learning to program in a new language requires practice, so here are a few additional exercises to help you practice. Spend time working on them, and follow the advice contained in this Chapter's **How to Build Code that Works** section. The more thought you put into solving the exercise problems, and then thinking through the implementation of the corresponding program, the better you will develop your understanding of the material in this Chapter.

Note

For the exercises below where there are snippets of code and you're asked to figure out the output you should be able to do this without compiling/running the code. This is where you verify that you understood the part of the material that the exercise is about. If you can not figure out the output without compiling/running the code, that is a sign you should probably go back and review the relevant section of these notes.

✍ **Exercise 2.8** Draw a memory diagram that illustrates what happens in memory when you run the following snippet of code:

```
int main()
{
    int x;
```

```
        int y;
        float pi;

        x=5;
        y=x+3;
        pi=y/2;

        printf("%f\n",pi);

        return 0;
    }
```

What will be the final value printed by the program?

✍ **Exercise 2.9** Consider the following snippet of code:

```
    float  convert_to_degrees(float angle)
    {
        // This function converts an angle given in radians to degrees
        return angle*360.0/(2.0*3.1415926535);
    }

    int main()
    {
        int x;
        float ang;

        x=2;
        ang=convert_to_degrees(x);
        printf("%f\n",ang);

        return 0;
    }
```

Draw a diagram that shows the memory model after we call **convert_to_degrees()** but **before the memory reserved for the function is released**.

What is the final value of **ang**?

✍ **Exercise 2.10** Consider the next little program:

```
    void hard_working_function(int x)
    {
        x=x*x;
        x=x/x;
        x=3*x;
        x=x+71;
        x=x/2;
    }

    int main()
    {
        int x;
        x=1;
        hard_working_function(x);
        printf("%d\n",x);

        return 0;
        }
```

What is the final value printed by the program?

✍ **Exercise 2.11** Write a program that

- Declares an array of 10 integer values
- Fills this array with equally spaced angles between 0 and 359 degrees
- Prints out the 10 angles both in degrees and in radians

✍ **Exercise 2.12** Remember that when we pass an array to a function, the function gets a pointer to the array (not a copy of the data), and recall that strings are arrays of chars. Now consider this program:

```
        void start_a_story(char input_string[1014])
        {
            char little_story[2048];

            strcpy(little_story,"Once upon a time...");
            strcat(little_story,input_string);
            printf("%s\n",little_story);
        }

        void main()
        {
            start_a_story("there was a blue rhinoceros named Randolph.");
        }
```

What is the output of this little program?

✍ **Exercise 2.13** Now consider the modified program from Exercise 2.12 shown below:

```
        char *start_a_story(char input_string[1014])
        {

            char little_story[2048];
            strcpy(little_story,"Once upon a time...");
            strcat(little_story,input_string);

            return little_story;
        }

        void main()
        {
            char *story;
            story=start_a_story("There was a blue rhinoceros named Randolph.");
            printf("%s\n",story);
        }
```

There is a major problem with the code above. What is it? (hint: when are boxes reserved for function calls?). Explain what the issue is, and what you expect would happen if we compile and run the code above.

✍ **Exercise 2.14** Let's consider the following little programs

```
        void main()
        {
            char little_string[1024];
            char *p=NULL;
            strcpy(little_string,"This is just a little harmless string");

            p=&little_string[0];

            printf("%c\n",*(p));
            printf("%c\n",*(p+3));

            p=p+5;
            *(p)='u';
            printf("%s\n",little_string);
        }
```

What is the output of the little program?

2.8 Building Programs That Work - Part 2

Writing programs that work, and writing good code, are habits that require work and discipline. Experience helps, so the more you practice, the better you will become at it. But it all begins by having a solid process for developing programs that solve a variety of problems. You will develop this process over the course of your career, and there are multiple disciplines that will contribute to it. For this introductory book, we will develop a skeleton on which you can later build a stronger, much more capable software development process.

Here is a short process that you should follow to solve any programming-related problems going forward.

Step 1 - Fully understand the problem you are expected to solve. In examples within the book, you will be working from a problem description of what your program should do, in more realistic contexts you may have a user with particular needs for what your software should accomplish. In every case, you **must fully understand the problem** as well as **any specified characteristics of the solution**. If the problem is being specified by a user, you will need to ask questions that help you fill-in any gaps in the original problem description.

Example 2.11 **Description:** You are asked to implement a program that will find all prime numbers between **2** and **N**, where **N** can be up to **10000**. The program should implement the method called **the sieve of Eratosthenes** to find these primes, and should print them out as a list separated by commas.

Understanding the problem: The description above is fairly short, and may not provide all of the information you need to solve this problem, but it does provide a good starting point. Reading the description carefully we note that:

- Our program will be tasked with finding **prime numbers** (what is a prime number?)
- These numbers have to be within a range of **[2,N]** where **N** can change but is at most **10000**
- To find the primes, we must use a method called **sieve of Eratosthenes** (what is that?)
- The primes that were found will be printed as a comma-separated list

The description of the problem was enough to give us the big picture, but now we need to fill-in the details. First, if we are not sure what a prime number is the first step is to learn about them and make sure that we understand

how to determine if a number is prime. We can't find prime numbers if we don't know how to check if a number is prime.

Secondly, if you, like me, have never heard of the **sieve of Eratosthenes**, we would have to go and find out what it is and how it works. This involves a bit of work, you can look this up online, or you can check it out in a book. The sieve works by **scanning the numbers that are in the specified range** and **progressively tagging** all numbers in this range that are multiples of the primes found so far. So, for instance, the first number scanned is **2** which is prime. The sieve then tags all multiples of **2** as not prime. The next number scanned is **3**, which was not tagged before so **it must be prime**, the sieve then tags all multiples of **3** in the range as not prime. Next up is **4** but it was tagged before because it's a multiple of **2** so it is not prime. Next is **5** which was not tagged previously so it must be prime and the sieve tags all multiples of 5, and so on.

By the time the sieve has scanned all numbers in the specified range, any non-tagged numbers left must be the primes.

Note that we have not said anything about programming, we haven't started implementing anything, we are just understanding what the problem is and what the task is that we need to solve. We should **not proceed** until we are certain we have **fully understood** the problem, the input our program will be required to work with, and the output it must produce, as well as any specified conditions on how the problem must be solved. Whenever you are not entirely sure about something, you should seek additional information. It is also possible you will come up with more questions as you work through step 2 of the process.

Step 2 - Think through the steps required to solve the problem *on paper*. You should **not start writing a program** until you have a well developed algorithm for what the solution should be. The solution should be detailed enough that you or anyone else implementing it can easily figure out what the different parts of the process are, how it can be broken up into functions, and how these functions will work together to produce the solution. You algorithm can be written in **pseudocode** or just as a list of steps, but it has to cover every important part of the solution.

Example 2.12 For the program that finds the prime numbers, given our understanding of the problem and the description of how the **sieve of Eratosthenes** works, we can come up with the sequence of steps that our program needs to carry out to solve the problem:

Requires: The value of **N** **Algorithm:**

- Declare an array with **N** integer entries. This array will be used to indicate if a number is prime or not. The value **1** will indicate a prime, **0** will indicate a non-prime
- Initialize all entries in the array to **1**. The process will then tag non-primes by setting their array entry to **0** as they are identified
- Loop over each value **i** from 2 to N
    - If the entry for **i** is **1**, then **i** is a **prime**. Loop through the array in increments of **i** tagging all multiples of **i** as non-primes by setting the corresponding array entry to **0**

- After the loop is complete, loop once again over the array entries from **2** to **N**, if the corresponding array entry has a value of **1** print the number

Note that the algorithm above states what information is **required** for the algorithm to work, and the description of steps is detailed enough that we could come up with a program that implements it. You should also note that **the algorithm is the solution**. Implementing the algorithm is really not the major issue when solving a problem - the key step is this one, figuring out what the algorithm is in enough detail that you can be confident that implementing that algorithm will solve the problem.

Needless to say, you can not expect a program to work if you skipped this step and simply tried to write some code without first thinking through the solution. That is a recipe for making a problem harder than it actually is: Writing a program from a complete algorithm is its own challenging task, but it is manageable. Similarly, solving a problem on paper is challenging but manageable. Trying to do both things at the same time is an easy way to turn two manageable tasks into a single unmanageable one.

Step 3 - Design the program that will implement your solution from Step 2. This means working out **how to break the algorithm into functions** that will carry out the required work. You need to think about **what data needs to be declared by each function**, and **how the different functions will communicate with each other** (input arguments and return values). You also need to **figure out the overall flow of the program**. You should be able to verify that your design has functions that take care of each of the steps in the algorithm you came up with, and you should be able to follow the program flow in your design and convince yourself it implements the algorithm correctly.

Example 2.13 For the prime finding program, and considering the algorithm proposed in Step 2, a reasonable design could look like so:

```
main()
    - Declares and owns the array of size N that will contain the 1s and 0s indicating
      which numbers are prime
    - Initializes this array to all 1s
    - Calls a function that implements the sieve of Erastosthenes on the array
    - Prints out the prime numbers after the sieve has completed its work

sieve()
    - Receives a pointer to the array with initially all 1s
    - Loops over entries starting from 2
      For each prime 'y' found, loop over the array tagging all multiples
      of 'y' as non-prime
```

For most problems and algorithms, there will be many possible designs, and there may be multiple reasonable ways to organize the program into functions. Later on in the course we will look at ways to decide whether one possible solution is better (in an objective way) than a competing one. For now, what matters is that you should come out of this step with a solid plan for your program, one which can be directly implemented.

Step 4 - Implement your solution. This is where you actually get to begin writing code. Notice that **by this point you have already solved the problem**, all that remains is to write code that carries out the algorithm, and implements the design that you already have. All your previous work will make implementing the actual program much easier, and will help you later on with testing and with (if necessary) debugging. Once you start writing the code you may find the need to adjust or change some of the details in your design, or your algorithm. However, you should not find that major changes are needed - if you do, that would mean your original solution was not properly thought out, or your design was not carefully planned out. In either case, **if you find a major issue while implementing the program you should go back and revise your solution and program design**.

Importantly: Hacking at code until it works is not a good process, it is not a good problems solving technique, and will not work for complex problems and reasonably long programs. So **avoid doing this**. Instead, make sure you have a solid algorithm to solve the problem, make sure your design is sound and reasonable, and the result of this will be code that is good **by design**.

Example 2.14 Given the design above, an implementation of the program to find and print prime numbers could look as shown below:

```
#include<stdio.h>
#include<stdlib.h>

#define N 1000

void sieve(int *p, int size)
{
    /* Implements the sieve of Eratosthenes on an array
    of the given size, it receives a pointer to the
    first entry in the array */

    int j,k;

    for (j=2; j<=N; j++)
    {
        if (*(p+j)==1)
        {
            for (k=j+j; k<=N; k=k+j)
            {
                *(p+k)=0;
            }
        }
    }

}

int main()
{
    int primeTags[N+1];
    int i;

    for (i=0; i<=N; i++)
    {
            primeTags[i]=1;
    }

    sieve(&primeTags[0],N);

    printf("The list of primes from 2 to %d is:\n",N);
    for (i=2; i<=N; i++)
            if (primeTags[i]==1)
```

```
                        printf("%d, ",i);
        printf("\n");

        return 0;
    }
```

As per the design, there are two functions: **main()**, and **sieve()**; **main()** declares and owns the integer array used to tag non-primes, and **sieve()** does the work of looping through the array tagging multiples of all primes found. The code follows directly from our design from Step 3, and did not require a large amount of work to write since the problem had already been solved, the algorithm had already been worked out, and the design had already determined how many functions there would be and what these functions would do.

Step 5 - Test your program. Of course, once you have written any program, you must ensure it works properly and the only way to do this is by **thorough testing**. Testing is a skill that requires time, practice, technique, and hard thought. **Testing is a topic that requires significant attention on its own so we will discuss it in detail in the next Chapter**. For now, we will simply **compile our code**, **deal with all compiler errors and warnings** as we learned in Chapter 1, and verify that **the program produces the correct list of primes** for a given value of N.

Compiling and running the code for **N=1000** as in the program above prints out:

```
./a.out
The list of primes from 2 to 1000 is:
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,
     101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191,
    193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283,
    293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401,
    409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509,
    521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631,
    641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751,
    757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877,
    881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997,
```

Should we be satisfied that the program didn't crash and printed out a bunch of numbers? **of course not!**. We have to check that **the program has correctly solved the problem**. Notice that this is often **very different from checking that the output is correct**.

A program can produce seemingly correct output, while having done the wrong thing internally. So whenever possible, we want to verify that the information computed or produced by the program is correct **internally**. Another way to think about this is: **we test for correctness, not output**.

In the case of this program, the task was to find the primes in the range **[2,N]**. From our program design we know the result of performing the sieve of Eratosthenes will be an integer array in which **primes will be indicated by a 1**, and **non-primes will be indicated by a 0**. Correctness of the results in this array requires us to check for two things:

- That all the numbers for which the corresponding entry is **1** are prime
- That all the numbers for which the corresponding entry is **0** are non-prime

We obviously do not want to do this by hand. Instead, we can write a test function, let's call it **check_result()**, that goes into the array after **sieve()** has done its work, and checks that both of the conditions noted above are true for the entire array.

Example 2.15 A sample function to check the correctness of the results produced by **sieve()** is shown below:

```
int check_result(int *p, int size)
{
    /* This function gets a pointer to the array with the results
    from sieve() and checks each entry starting from index 2.
    If the entry is a '1', it checks that the corresponding
    index is a prime number. If it's a '0', it checks that the
    number is not prime.

    If all entries in the array are correctly marked as prime
    or non-prime, it returns 1 (indicating everything is correct)
    else it returns a 0 (and also prints the index at which an
    erroneous value was found)
    */

    int i,k;
    int tst;

    for (i=2;i<=N;i++)            // For each entry in the array
    {
        tst=1;                    // Assume it is prime
        for (k=2;k<i;k++)         // Try to divide it exactly by ingeters
        {                         // from 2 to i-1
            if (i%k==0) tst=0;    // if we can do it, it's not prime
        }
        // Now we know if i is prime, check if it agrees with the array
        if (*(p+i)!=tst)
        {
            // Failed! something is inconsistent
            printf("Found a problem, integer %d is tagged with %d, prime test returns %d\n",
                i,*(p+i),tst);
            return 0;
        }
    }
    return 1;     // Everything looks good!
}
```

we also need to add a couple lines to **main()** to call this function and print a message depending on whether results check out or not (these lines would be added anywhere **after** the **sieve()** function has been called:

```
     // Call the test function to verify the result
    if (check_result(&primeTags[0],N))
    {
           printf("The results check out, everything appears correct\n");
    }
    else
    {
           printf("There was a problem, at least one entry was tagged erroneously by the
               sieve\n");
    }
```

compiling and running the code now gives the following output:

```
./a.out
The list of primes from 2 to 1000 is:
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,
    89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269,
     271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367,
    373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463,
     467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577,
    587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677,
     683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797,
    809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911,
     919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997,
The results check out, everything appears correct
```

so the test function has verified the work done by the **sieve()**, and no inconsistencies have been found. We can check that the test function will catch problems by **manually creating an inconsistency**. For example, we can add a couple lines of code that change some of the tags in the array to the wrong value - these lines are added **after** the call to **sieve()**:

```
primeTags[4]=1;   // Here we are saying that 4 is prime
primeTags[5]=0;   // and also that 5 is not prime
```

compiling and running the code now prints out:

```
./a.out
The list of primes from 2 to 1000 is:
2, 3, 4, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,
    89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269,
     271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367,
    373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463,
     467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577,
    587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677,
     683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797,
    809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911,
     919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997,
Found a problem, integer 4 is tagged with 1, prime test returns 0
There was a problem, at least one entry was tagged erroneously by the sieve
```

the test function correctly identified the first entry that was wrong in our array. If you look carefully, you can see that **4** was printed in the list of primes, and **5** was not. That may be hard to catch simply by inspecting the output, so please remember this: **write code to test your functions and program for correctness**, and **do not rely just on printed output** to decide your program is correct.

A final comment on the above: The example test shown above is a good start, but by no means does it constitute a full and comprehensive test for correctness for the program. We still don't know that the program works correctly **for different values of N**, or that **the output is actually printed correctly given the contents of the array**, or **that the code is actually implementing the sieve of Eratosthenes as specified in the problem description**. Testing is a thought and work intensive process, you should practice it as often and as thoroughly as you can, until you have built the ability to develop a sufficiently solid testing process for your programs that you can be highly confident that anything you ever make available for use is as likely to be completely correct as one could hope for. We will discuss testing at length in the next Chapter.

Chapter 3 Organizing, Storing, and Accessing Information

Now that we have a strong enough understanding of our programming language, we can start our exploration of the interesting problems and tools that are found throughout computer science. We will start by looking at the fundamental problem of **data and how to store it** so that we can **efficiently** find the information we need, and **manipulate** data as needed to solve whatever problem we are studying.

By **data** in this case, we usually mean **large amounts of information**. Organizing a few integers, or a couple strings, or some floating point numbers is not challenging, interesting, or ultimately useful. Where things get interesting is when we start looking at **large collections** of data items that our program needs to work with. A few examples of the kind of data items our programs may eventually need to work with include:

- Text documents - e.g. news articles, blog posts, web documents, pages from books, etc.
- Music files - or sound clips, podcasts, newscasts, etc.
- Pictures - collections of photos, digital archives, digital art collections, etc.
- Video files - movies, clips, recordings of work meetings, etc.
- Database records - for instance, the collection of information that represents each customer profile for a large online store. There are endless variations of these, and the amount of information available can be staggering.
- Results of computations - for example, the output of a program that does weather modelling will consist of huge amounts of numerical data that has to be stored, and then processed and visualized to enable humans to make sense of it.

In general, one of the first things you have to do when solving a problem using a computer is to carefully consider:

- What type of information we have to store, is it mostly numbers, text, a mixture of media, or something else?
- How much of it do we need to be able to handle - for example, your phone's photo app only needs to deal with the images you have stored in your device, Google on the other hand needs to be able to organize and search through billions of images available online, these are two very different problems.
- How the data will be accessed and used, and by whom - this will have implications for choosing how the information is organized and stored, and puts constraints on security measures we may need to put in place to restrict access to it.
- How to make the data easy to manage within a program - which is often different from making the information easy to access by a human.

In this part of the book, we will learn about **program-level organization and manipulation of data**. We will see how to use the simple data types provided by **C** in order to build richer, more useful, more flexible, and easy to use **data containers** that can be used to represent, store, organize, access, and manage pretty much anything you may wish to manipulate inside a computer.

The concepts and techniques covered here will be the foundation you will need later on in order to understand how to model information, and how the modeling of information affects the design of the software written to handle it (this is one of the main problems studied in Software Design). It is also the foundation on which databases are built. Nowadays, databases are needed almost for every application – from a cooking recipe app (which will have some form of searchable recipe database), to customer information systems for every kind of business both on-line and on-site. Databases are fascinating, so if you're curious about how they work you should follow up by taking a course or reading a book about them.

Note

Just how much data is out there? This is a simple question, but it doesn't have a particularly easy answer as it would appear **no one really knows exactly** just how much data is out there, or even **how to measure** what is out there: do we count only publicly-accessible data? - which leaves out huge collections of information available only to particular governments or corporations or organizations. Do we include data stored in user devices? - as opposed to only what is stored in some internet-reachable server and thus possible accessible by others. Do we include only **meaningful** data items? - that means, files and formats we can easily recognize and make sense of, which leaves out large chunks of data that is just numbers and that we can't understand unless we are provided with a thorough description of what it means.
However we decide to count, the unavoidable fact is that **there is a huge amount of information out there** and we should keep in our minds that programs and applications we write will, more often than not, need to deal with fairly large amounts of data. Here are a few facts about data that will help you wrap your head around just how much of it is stored out there.

- The **Google codebase** in 2016 included approximately **one billion files** and had a history of approximately 35 million commits spanning Google's entire 18-year existence. The repository contained **86TB of data** at the time, including approximately **two billion lines of code** in **nine million unique source files** (source: `https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext`). By 2024 the estimate is that the number of lines of code is in the tens-of-billions, but no official figure is available.
- **How much data is stored by Google?** Apparently the answer to this is *no-one (perhaps including Google) really knows*. But here's a fairly interesting analysis: `https://what-if.xkcd.com/63/` . The analysis provided states: *Let's assume Google has a storage capacity of 15 exabytes*, or 15,000,000,000,000,000,000 bytes. This is a not unreasonable estimate and was likely at the right **order of magnitude** for the actual storage available to Google when this post came out, several years ago. At the present time it has been suggested Google's total storage capacity may be in the **order of zettabytes (ZB)** - a zettabyte is 1000 exabytes, so 1,000,000,000,000,000,000,000 bytes.
- The number of pictures uploaded to facebook **each day** as of 2016 was closing on half-a-billion. According to `https://www.omnicoreagency.com/facebook-statistics/` *"350 million photos are uploaded every day, with 14.58 million photo uploads per hour, 243,000 photo uploads per minute, and 4,000 photo uploads per second."*. For Instagram, as of 2021, the figure was 95 million videos and pictures being uploaded daily. Also from this site `https://www.omnicoreagency.com/instagram-statistics/` we find that *"More than 50 Billion photos have been uploaded to Instagram so far."* and that *"Pizza is the most Instagrammed food globally, followed by Sushi."*

- From `https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#6a951bf860ba` we find that *"There are 2.5 quintillion bytes of data created each day"*, *"On average, Google now processes more than 40,000 searches EVERY second (3.5 billion searches per day)!"*, and *"We send 16 million text messages (every minute!)"*. This was in 2018, by now those numbers will have grown significantly.
- You should have a look here `https://everysecond.io/youtube` to get a real-time sense of just how much data is being produced and uploaded to some server every second of every day

These are just a few examples, but hopefully they drive home the point that storing, accessing, modifying, and maintaining a collection of information is **far from an easy or trivial problem**.

Figure 3.1: This is what the inside of a data center looks like. Somewhere in the world, in a data center similar to this one, a copy of this book is stored. *Photo: Global Access Point, Public Domain*

3.1 How to build a Bento Box

We know how to use the standard data types provided in **C**, however most interesting applications will require keeping track of data that is a bit more complex than a few integers, or floats, or even a few strings. The problem at hand is **how to design and implement a new data type**, something beyond what is already available in **C**, and that can represent **a much more interesting unit of information**.

As an analogy – a good meal is not composed of a single item like, broccoli (you should eat broccoli by the way, it's good for you!), but instead it consists of many different components put together in a way that makes a good meal. The individual components are ingredients you may find at any store, but the finished meal is much more interesting. If you have been to a Japanese restaurant, then you may have already seen a meal that is a great example of the process we are now going to apply to data types: **The Bento Box** (see Fig. 3.2).

Figure 3.2: Bento Box - the meal is composed of individual components, each in its own container, arranged to complement each other and each of them needed to complete the meal. *Photo: miheco - Flickr, CC - SA 2.0*

Our task here is to figure out how to represent **information about an item**, where this information is **more complex than what a single data type can hold**. For example, if we are going to write an application to store information about movies, we may need to store:

- The movie's title
- The year the movie came out
- The name of the director
- The name of the studio that produced it
- The review score it received on Rotten Tomatoes
- ... and possibly a lot more

There are several individual pieces of information that we need to keep track of for **each movie**, and each of them has **its own data type** – there may be strings, integers, floats, and so on. We will need to store information for **many movies** (it is not particularly useful if we can only store a few, remember we are learning how to deal with large collections of information).

Question: Given what we know at this point, how could we do this in **C**?

Using only **C's** standard data types, we would need to **create separate arrays** for **each of the different components** that make up a movie's information (from now on, we will refer to a single movie's information as a **movie record**):

- One array of strings for the movie titles
- One array of integers for the year when the movie came out

- One array of strings for the directors' names
- One array of strings for the studio
- One array of floats for the Rotten Tomatoes score

Now we can design a program that allows users to fill-in these arrays with information for each of the movies we'll keep in our app, and that provides the functionality the user wants.

Question: What do you think are the advantages and disadvantages of storing **movie records** in this way?

In practical terms, the implementation with arrays has a number of disadvantages that we need to be aware of:

- Because arrays in **C** have a **fixed size**, our application will be limited in the number of movie records that it can store and manipulate. We can run out of space and not be able to store any more movies, or, if we choose to make these arrays really huge to begin with, our app will be consuming a very large amount of memory most of which may go unused. This is wasteful and would cause the app to take resources that other programs may need.
- Because we are storing information in several separate **containers** (each array is a container), all of our code now needs to deal with multiple sources of data, and we must ensure that movie record information is kept **properly synchronized** across all of these arrays at all times. This adds complexity to the program, increases the likelihood of bugs, and makes testing of the app more complicated.
- **Conceptually** something is not right - a **movie record is a single unit of information** and yet we are working with it as if each part was its own thing. We have taken what should be a **bento box** and put each food item in a separate box with nothing to hold the separate components together to indicate they should form a single unit. This is not how we want to think of information, and it is not how we want programs to handle information. **Information that corresponds to a single item** (a movie in this case) **should be bundled together**.

To make the above more concrete, here is an example in **pseudocode** of what a program may need to do if we decide to use regular **C** arrays to store records, and we will compare that with a solution in which information is bundled together rather than spread into multiple containers.

Example 3.1

```
// Using one array for each piece of information that makes up a single movie record

// Here is what a function that adds a single movie to our app might look like
// (in pseudocode)

addMovie()
    Inputs:
            - The new movie's title
            - The new movie's year
            - The new movie's director
            - The new movie's studio
            - The new movie's Rotten Tomatoes score
            - The app's movie titles array
            - The app's movie years array
            - The app's movie directors array
            - The app's movie studios array
            - The app's movie scores array
            - The number of movies currently stored

    Returns:
            - The number of movies after we inserted the new one
              (it may not have changed if we were not able to add
               the new movie!)

    Procedure:

            Check that there is space left in the arrays
                if no space left, print an error message and
                return the current number of movies unchanged

            Fill in the new movie's title in the titles array
            Fill in the new movie's year in the movie years array
            Fill in the new movie's director in the directors array
            Fill in the new movie's studio in the studios array
            Fill in the new movie's RT score in the scores array

            Increment the number of movies in the database by 1
                and return the updated number of movies

// Here's what a function that deletes a movie from the app might look like (in pseudocode)
deleteMovie()
    Inputs:
            - The index of the movie we want to delete
            - The app's movie titles array
            - The app's movie years array
            - The app's movie directors array
            - The app's movie studios array
            - The app's movie scores array
            - The number of movies currently stored

    Returns:
            - The updated number of movies after we deleted the
              requested movie

    Procedure:

            Check that the requested movie index is valid (i.e. greater or
                equal to zero, and less than the number of movies
                currently stored)

                If the index is not valid, print an error message and
                return the current number of movies unchanged

            Deletion:
                If the index of the movie to delete is i, then starting
                with index i+1

                    Move all entries in the movie titles array one space back
                            (e.g. entry k will move to entry k-1)
                    Move all entries in the movie years array one space back
                    Move all entries in the movie directors array one space back
                    Move all entries in the movie studios array one space back
                    Move all entries in the movie scores array one space back

                Decrement the number of movies in the app by 1
                    and return the updated number of movies
```

Things you should note from the above example: Notice the large number of input arguments that each of our functions will require. Because we have multiple pieces of information, and multiple containers, the argument list for the functions will be unavoidably long. This makes the program more cumbersome to read and understand, and therefore harder to maintain. Importantly **if we need to add a new data item to a movie record**, for example, to store also the **movie's box office amount** (how much money the movie made), **we would need to modify the argument lists of all functions in the app that deal with movies**. That is not a great feature of our design using separate arrays. Also, notice that each function is now performing the same operation multiple times on different containers. The function that adds movies is adding information in multiple different places, the one that deletes movies is now performing a shifting operation on each of several arrays. This makes the code repetitive, cumbersome to maintain, harder to test, and it's easier for a programmer to inadvertently introduce bugs because repetitive code tends to **look right even when it is not**.

What would change if we could bundle information for each movie into a single **movie record** that contains (in a single place) all the different pieces of data that we need to store?

Example 3.2

```
// Given a data-type that can store all the information of a single movie as a single movie
// record

// Here's what the function that adds a movie to the app might look like (in pseudocode)

addMovie()
    Inputs:
            - The filled-in movie record for the new movie
            - The array that contains the movie records for all movies in the app
            - The number of movies currently stored

    Returns:
            - The updated number of movies

    Procedure:

            - Check that there is space left in the array that stores movie records
                if no space left, print an error message and return the current
                number of movies unchanged

            - Copy the new movie record onto the movie records array at the end
            - Increment the current number of movies by 1 and return it

// Here's what the function that deletes a movie from the app might look like
deleteMovie()
    Inputs:
            - The index of the movie we want to delete
            - The array that contains the movie records for all movies in the app
            - The number of movies currently stored

    Returns
            - The updated number of movies

    Procedure:
            - Check that the index is valid (greater or equal to zero, less than
               the current number of movies)
                    if the index is not valid, print an error message and return
                    the current number of movies unchanged

            Deletion:
                If the index of the movie to delete is i, then startint at i+1
                    Move all existing entries in the movie records array up by 1

                Decrement the number of movies stored by 1, and return the updated
                number of movies
```

This is already better even in pseudocode. Note that the argument list for the functions in the program is now much smaller and easier to understand. Now compare the body of the functions: With **bundled movie records** the functions are cleaner, shorter, easier to understand for someone reading through the procedure, and there is no duplication of work because everything happens in a single array rather than multiple ones. Additionally, if we decided we need to add the **box office total** for the movies, we would **not need to modify the function's argument list at all!** and the body of the functions would either **not change** or **change only in a minimal way**. This is a huge advantage in terms of building programs that are easier to understand, test, debug, maintain, and expand.

The example above motivates the need for a way to **bundle information together** so that **each data item** represents the totality of the information that we need to manage for **a single entity** (movies in the examples above) that our program will need to manage.

3.1.1 Compound Data Types (CDTs)

In **C**, we can define our own **compound data types (CDTs)** which are the programming equivalent of a Bento Box: They are composed of a set of simpler data types, each of which provides needed information about a single item or entity our program needs to handle.

Movies are fairly complex data items (if you look at the information on IMDB for a single movie you will find out all kinds of things). So, for the examples in this section we will use a much simpler example: Suppose we are writing a little app to keep track of restaurant reviews. Let's say we are going to call our app **Kelp**.

The fundamental unit of information we need to keep track of is a single restaurant review. A **restaurant review record** consists of:

- The restaurant's name (this is a string)
- The restaurant's address (this is also a string)
- The review score (let's say this is an integer in 1-5)

Here's how we would build a **compound data type** in **C** that could store all the information required for **one restaurant review record**:

```c
typedef struct Restaurant_Score
{
   char restaurant_name[1024];
   char restaurant_address[1024];
   int score;
} Review;
```

Let's have a look at how this declaration works, since **we will be using this throughout the book to declare our own data types**. The first line:

```
typedef struct Restaurant_Score
```

tell the compiler several things. First **typedef** tells the compiler we are **def**ining a new data **type**. The next part **struct Restaurant_Score** tells the compiler that our new data type is a **compound data type** (in **C** this is known as a **struct**), and that the name of the **struct** will be **Restaurant_Score**. The general form for defining a new **CDT** is **typedef struct [_struct_name_]**.

After the **typedef**, and encased by curly braces, is the list of **fields** (which are the individual data elements) that are bundled into the **CDT** we are creating. In the case above you can see the **restaurant name**, the **restaurant address**, and the **review score**; each with an appropriate data type. In the case of the two strings, we specified a length of **1024**. This is arbitrary, but should suffice to store any reasonable restaurant name and any reasonable address.

The last line, with the **closing curly bracket** is also important:

```
} Review;
```

This line completes the declaration of **the new CDT** by giving it a **name we can use to create variables of this type**. In this case, the name we have selected is **Review**. Please note that this is different from the **struct name** that was used in the very first line of the declaration for the new **CDT**. Now we can use the new **CDT** in a program and create reviews for restaurants just as we would create other **C** data types - let's look at an example:

Example 3.3

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define STR_LENGTH 1024    // A convenient constant to use
                           // in our program

// Here's our new CDT declaration, discussed above
typedef struct Restaurant_Score
{
    char restaurant_name[STR_LENGTH];
    char restaurant_address[STR_LENGTH];
    int score;

} Review;

int main(void)
{
    Review rev;    // Declaring one variable of type 'Review'

    // Let's assign values to the information in 'rev'
    // Individual components of a compound data type are accessed by using
    // the '.' operator:

    // Score is just an int, so we can do this:
    rev.score=4;
```

```
        // However, the address and name are strings, which means arrays. We
        // have to use a function from the string library to copy them over.
        strcpy(rev.restaurant_name,"Best Salads Ever");
        strcpy(rev.restaurant_address,"777 Wonderful Street");

        printf("This review has: name=%s, address=%s, score=%d\n",\
           rev.restaurant_name,rev.restaurant_address,rev.score);

        return 0;
    }
```

The program above is a very simple example of how we would declare and use variables that are of compound type. In the example, **main()** declares a single variable called **'rev'** which is of type **Review**. As we discussed just above, the **Review CDT** contains three fields needed to represent the information of a single **restaurant review record**; so our variable **'rev'** has **3 fields**, and each individual field can be accessed as needed by using the **'.'** operator. In Example 3.3, all that we do is assign values to the different fields of the **CDT**, and then print out the information we just stored there.

Question: What happens in the memory model when we declare a variable that is of compound type? Recall that by defining **CDTs** we wanted to be able to **bundle** information so that we could treat **each record** for a complex entity (like movies or restaurant reviews) as **a single box that contains all the things we need**. So, in the memory model **we will represent CDT variables as a single box** which contains space for the different parts that comprise the **CDT**. This is illustrated in Fig. 3.3

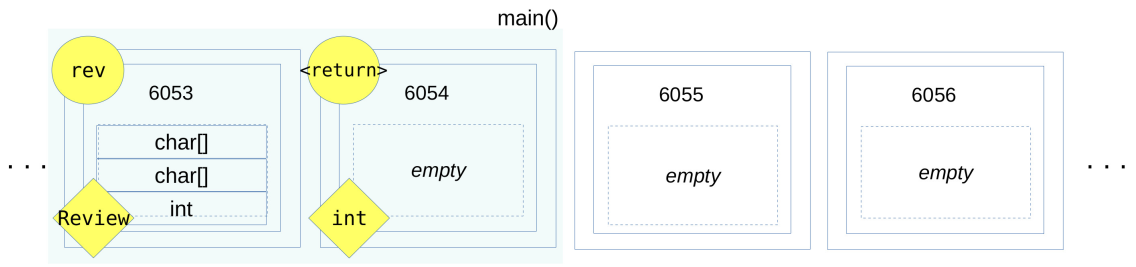


Figure 3.3: Memory model for the program in Exercise 3.3. Note that there is only **2 boxes**, one for the **Review** variable called **'rev'** which is of compound type, and one for **main()'s return value**

Note

Remember this: Variables that are **compound type** correspond to **individual units of meaning** in our program. So we treat them as **a single box** in the memory model. Importantly **this is a conceptual representation, part of our memory model's abstractions** so we can think about how information moves around in our programs in the correct way. How the pieces of information inside the **CDT** are represented, stored, and organized in the actual computer memory is a lot more complicated - but happily we do not need to worry about that, the compiler takes care of everything so we can use **CDTs** as bento boxes, with each of the parts we specified always bundled together.

Compiling and running the code above produces

```
>./a.out
This review has: name=Best Salads Ever, address=777 Wonderful Street, score=4
```

One very nice thing about **CDTs** is that once we have created them, we can use them just like we would use any other of the standard data types supported by the language. Here are some of the things we can do with CDTs:

```
    Review rev1, rev2;        // Declare individual variables of this type whenever we like
    Review many_reviews[100]; // Create arrays in which each entry is a CDT
    Review *rp;               // Declare and use pointers to access information in CDTs

    rev2=rev1;                // We can copy CDT variables onto each other, this would
                              // copy the values in each of the fields in 'rev1' onto
                              // the corresponding field in 'rev2'

    rev1.score=4;             // We can access individual fields in a CDT by using the
                              // '.' operator
    rp=&rev2;                 // We can get the address (locker number) of a CDT variable
                              // and store it in a pointer

    rp->score=3;              // And we can easily use a pointer to access information
                              // in a CDT variable by using the '->' operator. This
                              // particular instruction is exactly equivalent to
                              // rev2.score=3;
```

We will practice all of the above as we look at how to **pass CDT variables into and out of functions**.

3.1.2 Passing compound types between functions

Like any other variable, we will find we need to **pass CDTs** as input arguments to functions, and/or to **return a CDT** from a function. The way this is done is identical to the way we pass standard **C**-types between functions. For example, let's define a very short function that updates the **score** for a restaurant review

```
    Review change_score(Review old_rev, int new_score)
    {
        old_rev.score=new_score;
        return old_rev;
    }
```

This function takes as an arguments a CDT of type **Review** called **'old_rev'**, an **int** called **'new_score'**; and **returns** a CDT also of type **Review**. Let's now see an example of how we would use this function, and **what happens in memory** when we call **change_score()**

Example 3.4

```
        int main()
        {
            Review rev1, rev2;

            strcpy(rev1.restaurant_name,"The Home of Sushi");
            strcpy(rev1.restaurant_address,"555 Ellesmeadow Rd.");
            rev1.score=3;

            rev2=rev1;

            rev2=change_score(rev2,4);
        }
```

The first thing the program does is create two CDT variables to hold restaurant reviews (Fig. 3.4). As expected, this creates **two boxes** each of which contains space for the different parts of each of the reviews.

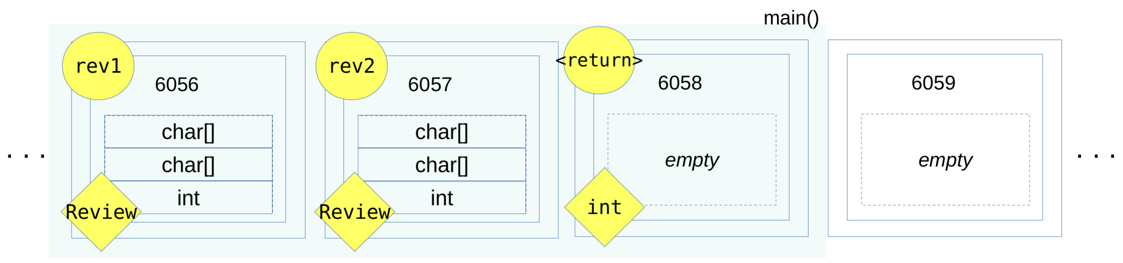


Figure 3.4: Memory model for the program in Example 3.4 just after memory has been reserved for **main()**.

The next three lines fill-in the information in **'rev1'**, which in the memory model looks as shown in Fig. 3.5.

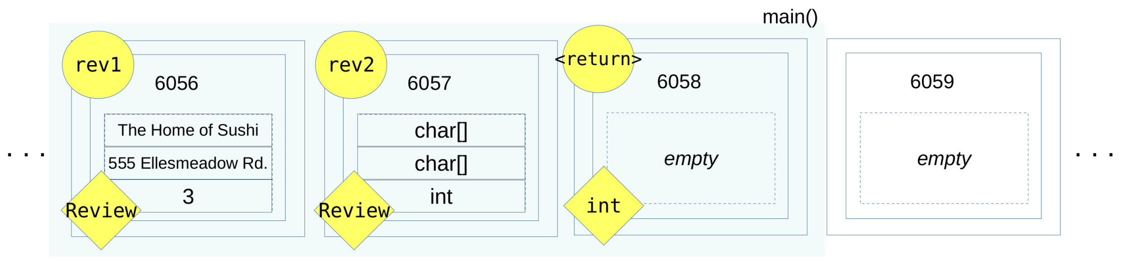


Figure 3.5: Memory model for the program in Example 3.4 after filling-in information for **rev1**.

The line **'rev2=rev1;'** makes a **complete copy** of the contents of the box tagged **'rev1'** onto the box tagged **'rev2'**. This means that each field is duplicated exactly. Needless to say, both boxes have to be exactly of exactly the same type (**Review**) for this to work. The result is shown in Fig. 3.6.

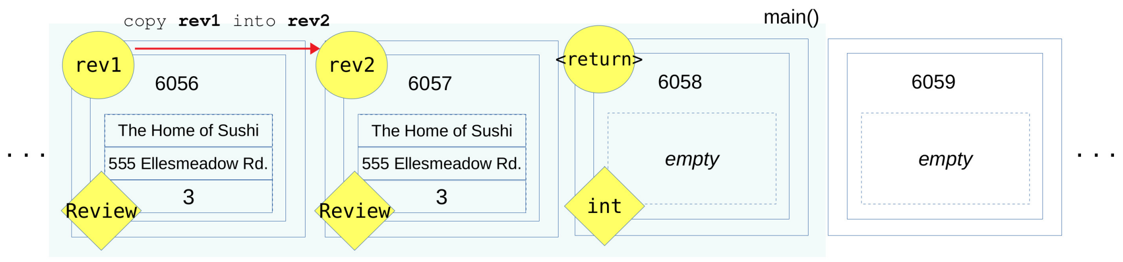


Figure 3.6: Memory model for the program in Example 3.4 after the instruction **'rev2=rev1;'**.

At this point we have two identical boxes, each containing the same address, restaurant name, and score. With the instruction **'rev2=change_score(rev2,4);'** several things happen. First, space is reserved for the function's input arguments and return value. The input arguments are one variable of type **Review** called **'old_rev'**, and an integer variable called **'new_score'**. Additionally space is reserved for the return value which is of type **Review**. As part of the process, **the value of the CDT** *rev2* **being passed to the function is copied into the function's** *old_rev* **argument**, and the value **4** is copied into the function's argument **'new_score'**. The situation is shown in Fig. 3.7.

The essential fact to keep in mind here is that **passing a CDT into a function** involves **making a copy of the CDT** into the corresponding input argument. Compound data types are **passed by value** just like regular data types

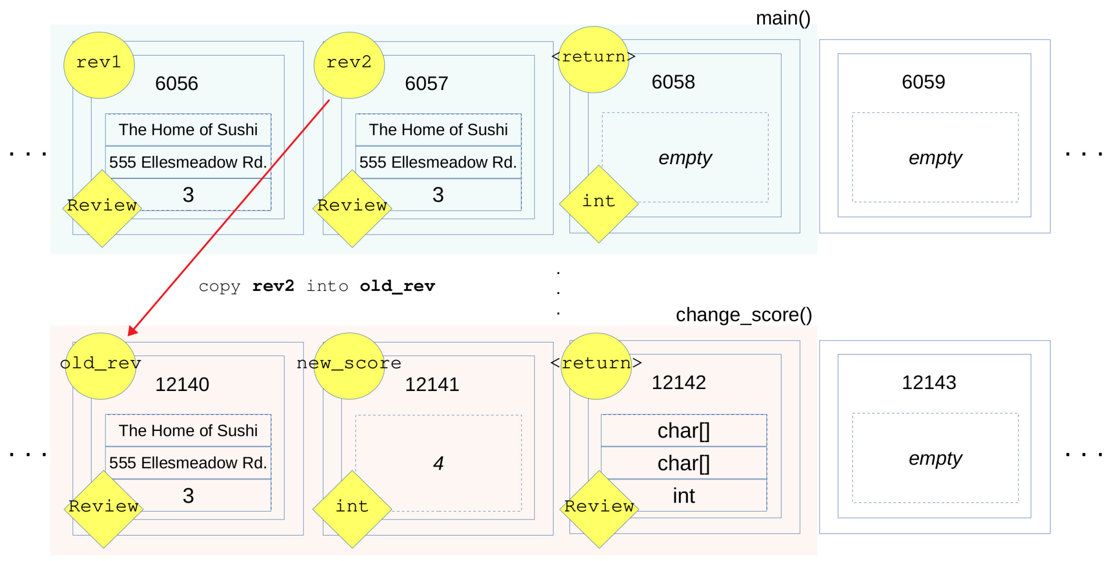


Figure 3.7: Memory model for the program in Example 3.4 after the instruction **'rev2=new_score(rev2,4);'** has reserved space for the function's input arguments and return value, and copied the input arguments into their corresponding boxes.

int, float, and **char**.

Once the function's memory has been reserved and the input arguments have been set up, the program continues with the instructions inside the body of the function. This very simple function only updates the score for the input movie review. Note that **the score being change is in the** *old_rev* **CDT**. Nothing has changed in the original variable **'rev2'** which belongs to **main()** and which we intended to update by calling this function. This is shown in Fig. 3.8

Finally, the **'return old_rev;'** statement completes the function call and does the following: First, **the value of** *old_rev* **is copied onto the function's return value** box (because we said that is the variable we want to return), and then the function returns control to **main()**. At that point, the **return value** of the function is copied onto **rev2** as directed by the original instruction **'rev2=change_score(rev2,4);'**. Only at this point is the score in **'rev2'** actually updated. This is illustrated in Fig. 3.9.

Once the **return** statement has done its work, the space reserved for **change_score()** is released.

The above process seems unnecessarily tedious, but it is important to see it at least once in detail so as to properly understand that

- **CDTs** are treated just as regular data types, and are **passed by value** (by making a copy) into and out of functions.
- The only thing that changes is the amount of information being copied.
- The information inside a **CDT** is always treated as a bundle, and the entire bundle is copied when necessary. The fields inside the **CDT** are never split from the bundle or left behind when the **CDT** is passed around the program.
- If the **CDT** contains a large amount of information and many different fields, the process of making copies of it could cause significant **overhead** (i.e. it can be slow). This can be important if many function calls are

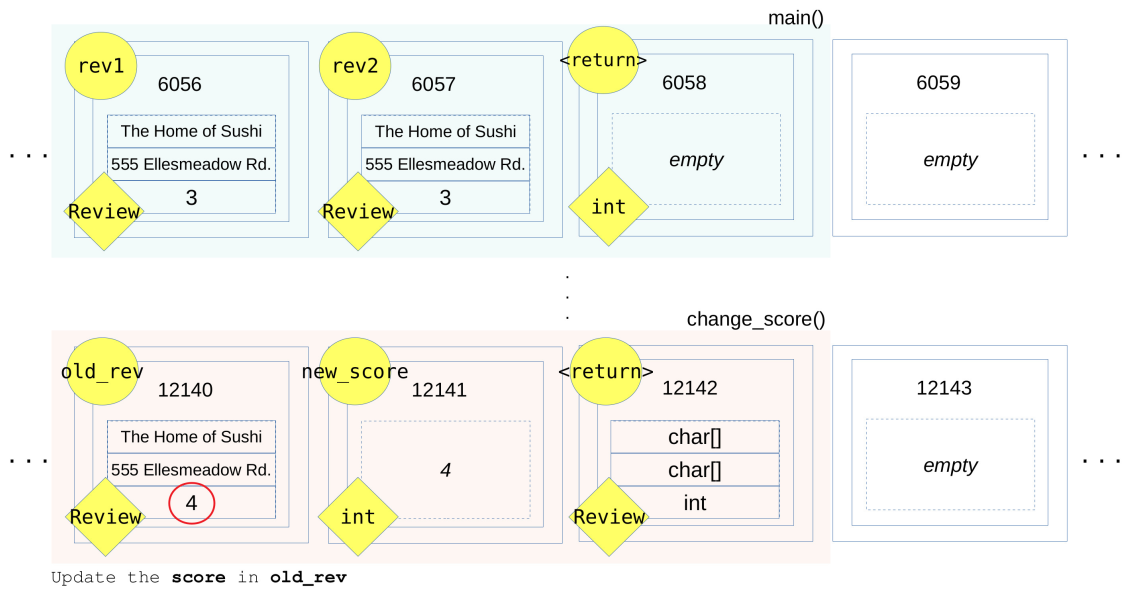


Figure 3.8: Memory model for the program in Example 3.4 after updating the score for the **old_rev** CDT.

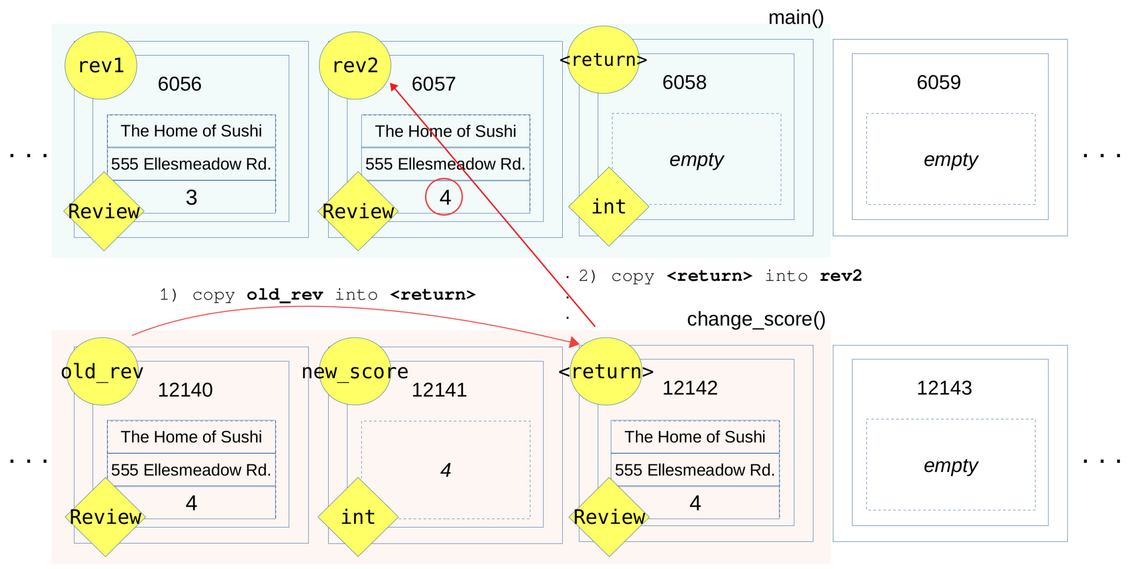


Figure 3.9: Memory model for the program in Example 3.4 showing the effect of the function's **return** statement.

required to get work done.

- To avoid the **overhead** of copying large **CDTs**, we often prefer to use **pointers** to access and modify information stored inside Compound Data Types. This is the same reason we don't make copies of **arrays**, and instead use **pointers** to access and modify array contents.

3.1.3 Using pointers with Compound Data Types

As we just saw, passing compound types around could involve a large amount of duplication of information. Much like arrays, what we often need is a way for a function to directly access and if necessary change the contents of a **CDT** defined outside. Just like with arrays, the way to do this is through the use of pointers. Let's see how we use pointers to handle compound data types:

Example 3.5

```
int main()
{
   Review rev;
   Review *rp=NULL;     // Remember to always initialize pointers to NULL when
                        // you declare them!

   strcpy(rev.restaurant_name,"The Baking Sleuth");
   strcpy(rev.restaurant_address,"221B Baker Street");
   rev.score=5;

   rp=&rev;             // Get the address of 'rev' and store it in our pointer

   rp->score=4;         // Use the pointer to change data stored in 'rev'

   return 0;
}
```

The code above declares one variable of type **Review** called **'rev'**, and a pointer **'rp'** to a variable of type **Review**. The next couple of lines fill-in the information for the review. The next line **'rp=&rev;'** is read as **take the address of *rev* and store it in pointer *rp***. Thereafter, **'rp'** contains the address of our review variable, and we can use the pointer to access and modify the information contained in that review. In our memory model, this would look as shown in Fig. 3.10.

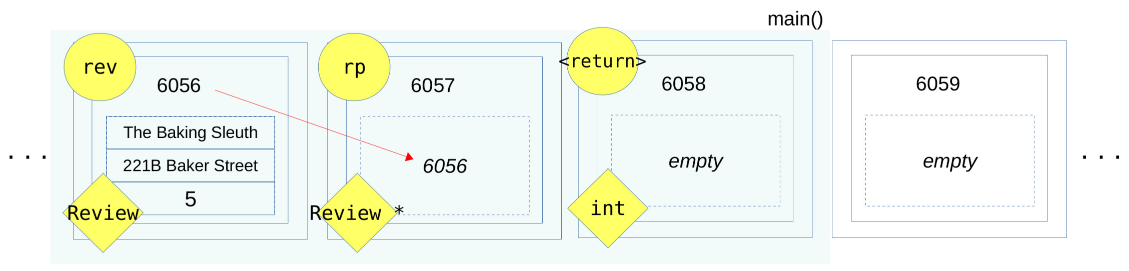


Figure 3.10: Memory model for the program in Example 3.5 after the line **'rp=&rev;'**.

Remember that to access the different **fields of a CDT** we use the **'.'** operator. This works only for **variables** of that type. With **pointers**, we use the **'->'** (arrow) operator instead:

```
rp->score=4;
```

This line updates the **score** field in our CDT variable **'rev'** to **4**. The nice thing about using pointers to access information stored in **CDTs** is that the syntax is much easier to manage (remember that with regular type variables, we need to use the **'*'** operator, which can lead to clunky and hard to read code). With **pointers to CDTs** we use the arrow operator **'->'** to select fields we want to access and/or update. Say, for instance, that we wanted to update the address of the restaurant, we could do so by using the pointer with the following line of code:

```
strcpy(rp->restaurant_address,"222A Baker Street");
```

Let's see how things change in the memory model if we take the program from Example 3.4, and modify the **change_score()** function to use pointers.

Example 3.6

```
void change_score(Review *rp, int new_score)
{
   rp->score=new_score;
}

int main()
{
   Review rev1, rev2;

   strcpy(rev1.restaurant_name,"The Home of Sushi");
   strcpy(rev1.restaurant_address,"555 Ellesmeadow Rd.");
   rev1.score=3;

   rev2=rev1;

   change_score(&rev2,4);
}
```

The first thing to note in Example 3.6 is the declaration for the function **'void change_score(Review *rp, int new_score)'**. It now takes as input argument **a pointer to a *Review* type variable** instead of **a copy of a *Review* variable** (as it did in Example 3.4). It also has **no return value** since this time around the function will directly update information stored in the **CDT** variable owned by **main()**.

Let's look at the memory model at the point where the program calls **change_score()** and the function updates the review score. The compiler will reserve two boxes for function **change_score()**, the first one is a pointer to a **Review** type variable, and the second one is an **int** (there is no return value, so no box is reserved for that). The function itself consists of a single line **'rp->score=new_score;'** which **directly updates the score in variable *rev2***. The situation in memory after the score is updated is shown in Fig. 3.11.

Compare the memory model in Fig. 3.11 against the original one in Fig. 3.9, and you can see that

- If we use a pointer, **there is no duplication of information** - the contents of **rev2** are never copied into the function. The version without pointers had to make a copy of the entire **CDT** three times.
- The memory model for the function with pointers is cleaner and easier to understand.

Hopefully this shows that even with a small **CDT** and a simple program, there are definite advantages to **using pointers to access and modify information stored in CDT variables**. This is one of the main reasons we will be using pointers throughout most of the programs we will write for the rest of the book.

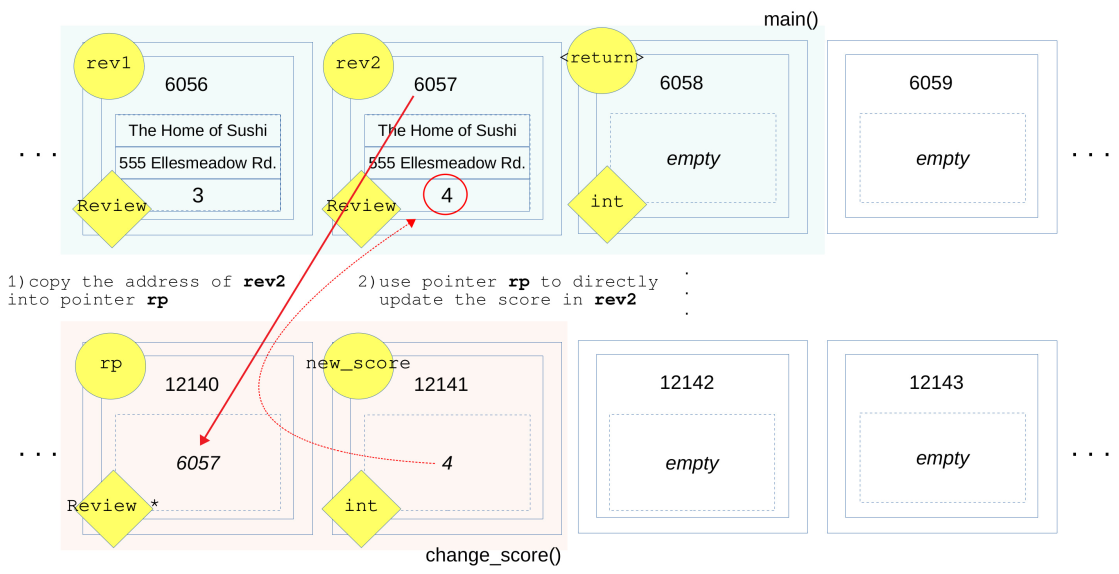


Figure 3.11: Memory model for the program in Example 3.6 after the call to **change_score()** has been set up.

3.2 Getting user input

From this point onward, we will be working with information units that are more interesting, and writing programs that handle larger amounts of information. As a result we will often need to request input from the user. Let's see how to do that in **C**.

3.2.1 Numeric types - integers and floats

For numeric data, we use the **scanf()** function. This function takes a **formatting string** that determines how the user's input is going to be converted into values that can be assigned to variables in our program. The formatting string uses the same format specifiers as **printf()**. Let's see an example:

Example 3.7

```
#include<stdio.h>

int main()
{
    int x,y;
    float pi;

    printf("Enter two integer numbers and one float on the same line\n");
    printf("Separated by spaces\n");

    scanf("%d %d %f",&x,&y,&pi);
    getchar();

    printf("Read: %d, %d, %f\n",x,y,pi);

    return 0;
}
```

Compiling and running the code above results in:

```
>./a.out
Enter two integer numbers and one float on the same line
Separated by spaces
2 3 1.2
Read: 2, 3, 1.200000
```

Things to note:

- The formatting string for **scanf()** specified that we want **'%d, %d, %f'**, so, one **int**, one **int**, and one **float**. Whatever the user inputs will be interpreted as values to be assigned to these data types, in the order specified by the formatting string. Remember that **scanf()** does **not validate input**. If the user inputs anything other than the expected data types, the resulting values will be **junk**.
- Because we want to read multiple values with one call to **scanf()**, we can not rely on the return value of **scanf()** to get our information. Instead, **scanf() takes in pointers to the variables where we want to store the information** the user provided. The pointers have to correspond to **variables of the correct data type**, and **in the exact same order as specified by the formatting string**. If we try to store information into the wrong data type, the result will be **junk**.
- There is a call to **getchar()** just after **scanf()** because **scanf()** will ignore the **[enter]** key the user pressed after inputting values. If we don't remove it, it will mess with any further input that our program requires.

To illustrate the point that **scanf()** does not provide any checking for what the user inputs, or whether it matches the data types we expected, see what happens when we run the same program but the user doesn't type-in the requested information and instead inputs gibberish.

```
>./a.out
Enter two integer numbers and one float on the same line
Separated by spaces
ahsga tsafhsgah 3
Read: 21893, 408739536, 0.000000
```

That clearly makes no sense. **You should always check that the user input is reasonable before using it in your program**. Checking that the input is reasonable is called **input sanitization** and involves setting reasonable bounds on what the input values should be. For example, if we are reading a score for a restaurant review, and we know that scores are in 1 to 5, we can check that the score read from the terminal is valid, and if not, ask the user to input a valid score.

✍ **Exercise 3.1** Write a little program that declares **an int array** with 10 entries, it **asks the user** for the values for each of these entries (these values should be in 0 to 100); and then **computes and prints out the average of the values in the array** (in effect, you're implementing the **average()** function found in most spread-sheet applications!)

3.2.2 Reading strings from the terminal

We **can not use scanf() to read strings** because **scanf() interprets spaces as delimiters**. Every space in the input string would be taken to mean that a new value for a separate variable is being provided. Instead, we will use a different library function called **fgets()** (the name comes from **GET S**tring).

Here's how you use **fgets()** to read strings from the terminal:

Example 3.8

```
#include<stdio.h>

int main()
{
    char my_string[1024];

    printf("Please type one string\n");
    fgets(my_string, 1024, stdin);

    printf("The input string is: %s\n",my_string);
    return 0;
}
```

The only new thing here is the call to **fgets()**

```
fgets(my_string, 1024, stdin);
//                      ^------- This specifies we want to read from the STanDard INput
//                                (the terminal)
//                ^------------ This is the maximum length of the string we expect to read
//       ^---------------------- And this is the name of the string where we will store
//                               user input
```

Note that **the maximum length we specify** for **fgets()** must be **less than, or equal to the length of the char array** where we are storing the user input. In reality, if we specified that the maximum length is **N**, then **fgets()** will read at most **N-1** characters from the user's input, because we need to reserve one character in order to store the **end-of-string** delimiter **'\0'**. While for many of our programs we will be reading input from **stdin**, the function **fgets()** can be used to read from other sources of data, such as files, network sockets, etc.

Compiling and running the program above produces:

```
> ./a.out
Please type one string
Here is one string the user typed-in!
The input string is: Here is one string the user typed-in!
```

Note

Be careful your string arrays are large enough to contain the information you will be reading, and use **fgets()** carefully. Trying to store a string in an array that is too small for it will crash your program.

To make the above point more clear, here is what happens when we change the program in Example 3.8 so that the **my_string** array has a size of only **10 chars**, and then we compile and run the code again and let the user type-in what they want:

```
$ ./a.out
Please type one string
This string will not fit within a 10 entry array, something bad may happen!
The input string is: This string will not fit within a 10 entry array, something bad may happen
  !
```

```
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

The specific way the program crashes (remember that with **array indexing**, or **pointer+offset** problems the result may be different on different computers, operating systems, or even the same computer at different times) is not important. What is important is for you to remember that **you can not control what, or how much the user will type**, so your program has to be written in such a way that whatever the user types-in you will not get in trouble. This includes **input sanitization** as discussed above for **numeric data types**, as well as **careful use of string arrays** and **making sure that fgets() never reads more than can fit in them**.

3.2.3 Practice Exercises

Let's take a moment to practice what we have learned up to this point regarding **CDTs** and **reading user input**. Take some time to try solving the following exercises **before** moving on to the next Section.

✍ **Exercise 3.2** Write a small program that:

- Declares a **CDT** that can store restaurant reviews. The **CDT** must include the three fields we used above in the examples, but also include **the average cost of a meal** at the restaurant, and **whether or not it does delivery**. You are free to decide **what data types** to use for the new fields.
- Declares one or two **Review** variables in **main()** so we can store information for one or two restaurants.
- Has a function called **fill_in_review()** that receives the information for each of the fields of a single review (as separate data items), and fills-in the corresponding values in a **Review CDT**. The function **must use pointers** to access/modify information in the **CDT** without making unnecessary copies.
- Calls the **fill_in_review()** function to fill-in the review variable(s) in **main()**.
- Prints out the information stored in the review(s).

✍ **Exercise 3.3** Modify the program from Exercise 3.2 so that

- It declares **an array for 10 Review CDTs**.
- It uses the function **fill_in_reviews()** to fill-in the information for each of 10 restaurants.
- The function **fill_in_review()** asks the user to input the information it needs for an individual restaurant, and reads that information from the terminal.
- It prints out the review information for the 10 restaurants.

3.3 Handling realistic amounts of data

At this point we know how to create custom boxes to store information, and it is time to turn our attention to one of the fundamental ideas this book is about. To understand what we're going to do, let's think a bit about what would happen if we wanted to implement the restaurant review app using only what we know up to this point:

- We know how to implement a new **CDT** to store information about reviews
- We know how to declare and use **Review** type variables

- We know how to pass reviews between functions in our program, both by copying them, and by using pointers
- We know how to get input from the user to fill-in a review's data

Question: How would our program be able to store multiple reviews **in a way that makes the information easy to access/modify**?

Suppose we say we want to use an array (so far this is the only **container** we know for storing multiple items of a given data type in **C**); so we go ahead and declare:

```
Review all_reviews[100];
```

This would reserve space for **100** reviews, they would be stored in consecutive boxes in memory (as is always the case with arrays of any data type), and they would be easily accessible to our program.

However, as was noted earlier in the Chapter, the **fixed size** of the array becomes a limitation, further, if later on we decide to change the size of the array, we will need to go through the entire program and change the size wherever it is being used. This makes maintaining the program more time consuming and increases the likelihood there will be a bug introduced over time when we forget to change the size somewhere.

But suppose we decide to be a bit smarter, and we do the following:

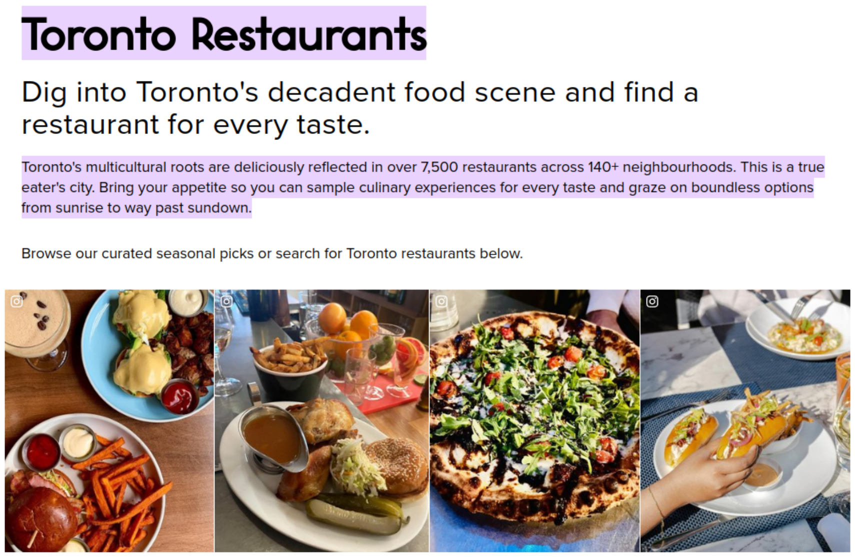
```
  // At the top of the program, we have
  #define    MAX_REVIEWS  100000

  // Then later on (maybe in main())
  Review all_reviews[MAX_REVIEWS]

  // Similarly, anywhere the program needs to
  // use the array size, it can simply use
  // MAX_REVIEWS. Changing the size of the
  // array becomes easy
```

The version above is better in that we now have a very large array, we're unlikely to run out of space at least for a while, and changing the size of the array can be easily accomplished by changing the definition of **'MAX_REVIEWS'**. However, the array **may be mostly empty** for a good part of the time. Because space for arrays is **reserved all at the same time**, the program will obtain one hundred thousand boxes for restaurant reviews, and keep them around even if it is using only a much smaller number of them to actually store information.

For example, as of **2024**, there are approximately **7,500 restaurants** in the city of **Toronto** (see Fig. 3.12). This means that for even a large city with a wide variety of places to eat, declaring the **all_reviews** array to have **100,000** entries will result in a significant waste of space. Over **90%** of the array will be empty. Even if we consider future growth, it seems **100000** may be too big. However, if we instead think about the total number of restaurants in **Canada**, we find that the number may be somewhere between **70,000** and **100,000** and our array will barely fit them all.

What we should remember from the above

- Arrays are wonderfully useful when we have a **known amount of data to work with**, and need a simple, easy to use **container** for storing and managing the data. They are commonly used in data processing applications to represent and manipulate numeric data.



Toronto Restaurants

Dig into Toronto's decadent food scene and find a restaurant for every taste.

Toronto's multicultural roots are deliciously reflected in over 7,500 restaurants across 140+ neighbourhoods. This is a true eater's city. Bring your appetite so you can sample culinary experiences for every taste and graze on boundless options from sunrise to way past sundown.

Browse our curated seasonal picks or search for Toronto restaurants below.

Figure 3.12: Toronto is a good place to be, if you like food!. *Image from: https://www.destinationtoronto.com/restaurants/*

- Because they have **fixed size**, they could end up being too small to store the information we need. Conversely if we define them to be very large from the start, they will likely waste a lot of space that may or may not ever be used.
- Because of these limitations, they are not the right tool for **implementing an information storage/retrieval system** that is intended to work on a **large collection** of data whose size is both **changing constantly**, and not **known inadvance**.

Here's the problem we would like to solve:

We need to develop a way to

- **Store, keep organized, and update** a **large collection of items** that represents the data our program will manage (e.g. the collection of reviews in our restaurant reviewing app).
- Our solution should allow us to **keep as few or as many instances of individual data items as we need**. We don't know the number in advance, and it may change over time. We **don't want to be constrained by a fixed number of items**.
- Space should be **reserved on-demand** as new data items are added to our collection. This is to **avoid wasting computer storage** by pre-reserving large amounts of space. In other words, our storage solution should be extendible.
- Our solution should enable us to **search for, access, modify, and delete** any individual data item in our collection.

The above is a fairly general description of what a **database** is. Indeed, the definition of what a **database** is, directly from **Oracle** reads: **A database is an organized collection of structured information, or data, typically stored electronically in a computer system.**

Of course a modern-day database is incredibly complex and very powerful. In what remains of the Chapter we will start exploring the ideas, principles, tools, and problems that arise when we consider the problem of **maintaining an organized collection of structured information**.

3.4 Containers and Lists

A **container** is a construct (something we have built) that provides **a means for storing, organizing, and accessing a collection** of data items of a given type. Notice that this is a very general definition - it doesn't specify how the data will be organized, it doesn't specify how the data will be stored in memory (or on a hard drive if we want to make a persistent copy), and it doesn't say how we will implement functions to access and modify data items in the collection.

An **array is a very simple but limited container**. We have discussed above the limitations that encourage us to develop a better solution for storing data when we don't know in advance how much of it our program will have to handle.

Let's have a look now at what is possibly the simplest container that:

- Allows us to keep a collection of data items
- The collection size **can grow or shrink over time**
- The data is organized in a simple, easy to understand way that allows us to find what we need

The container we are talking about is called a **List**, and its main property is that **The data items are stored in sequential order, one after the other, and for every item we can tell what the next item is (if there is one)**.

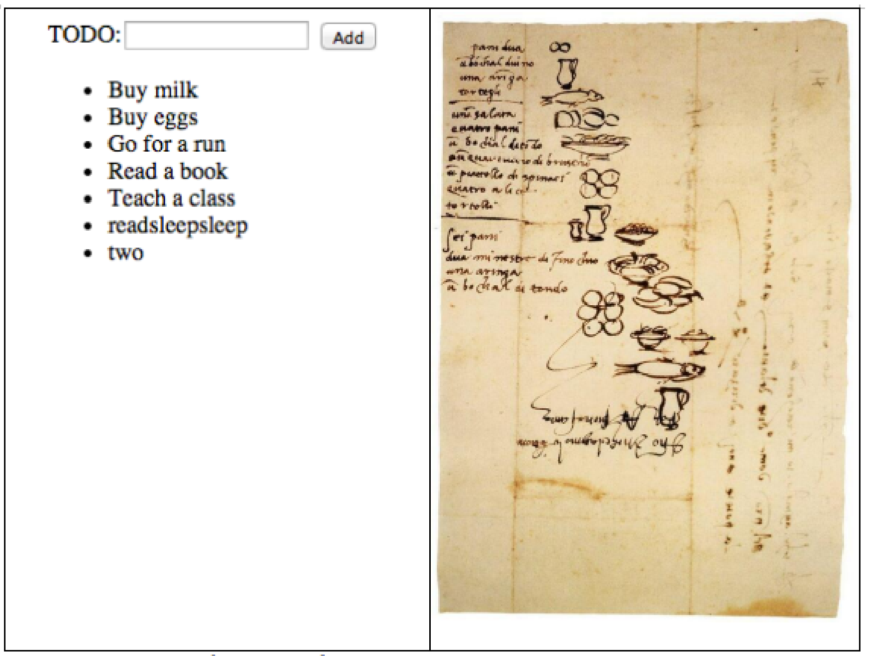
This is also a fairly general definition - **on purpose!** The goal of this definition is to provide only the basic properties of a list in a way that applies to any implementations you could write for it. This is important because it doesn't matter how you implement the list, or in what programming language, or what type of data it contains, a list is still a list.

To make the point perfectly clear, in Fig. 3.13 you can see two very different implementations of lists - hand-written vs. computer-made, shopping list vs. to-do list, and Italian language vs. English language. The details of how the list was created, or what it contains, do not matter. They both share the same key properties of storing a collection of items, sequentially ordered, and so they both are examples of a list.

3.4.1 List Abstract Data Type (List ADT)

The **List Abstract Data Type** extends our definition of a list container by specifying the operations that the list must provide. That is, in addition to representing a collection of data items that are sequentially ordered, the List ADT requires the following operations to be implemented:

- Creating a new (empty) list
- Adding items to an existing list
- Removing items from a list
- Searching for a specific data item


TODO: Add

- Buy milk
- Buy eggs
- Go for a run
- Read a book
- Teach a class
- readsleepsleep
- two

Figure 3.13: Two examples of lists. On the left we have a to-do list created with an app. On the right we have Michelangelo's shopping list from the 16th century. *Images: (left) Lubaouchan, Wikimedia Commons, CC-SA 4.0; (right) Michelangelo, Wikimedia Commons, Public Domain.*

The **search** operation is needed so we can find, modify, and view the contents of specific items in our collection. For example, in our restaurant reviews app, we may want to update the score for a restaurant already in the list. **Search** is also needed to check whether a specific item is already in the collection (i.e. we need it in order to avoid duplicating information).

There are variations on the definition above. We may find versions of the **List ADT** that include other operations, for example, **getting the length of the list**, or **inserting items at specific positions** in the list (common options include at the front vs. the end). The definition we provide here contains the fundamental operations we will find on pretty much any list we will encounter in the future.

3.4.2 Why is this called an 'abstract' data type?

This is a particularly important point: The List ADT we defined above **is called abstract** because **it does not specify how the List ADT and its operations are to be implemented**. There are many possible ways in which we could build our list, and we could implement it in any programming language we know of. **Two actual implementations of the List ADT could be completely different from one another**, and yet, anyone who knows what the List ADT includes will know to expect **a collection of data items, sequentially ordered, that supports declaring a new list, adding and deleting items, and search.**

This is useful because it means that once you know how and when to use a **List ADT** to store and organize data, **you can do so using any of the implementations of the ADT**, in any programming language, **without having to worry about the implementation details.**

Note

Abstract Data Types are a fundamental component of problem solving in computer science - they allow us to think in terms of how data is organized, and what operations can be performed on that data, so we can determine the optimal way to store and manage the information for the specific problem we need to solve without having to worry about implementation details.

3.5 Linked Lists

One of the most common implementations of the **List ADT** is **the linked list**. To understand how a linked list works, we can turn back to our original analogy of memory being just a very large room full of numbered lockers.

Here's a real-world example of the process we follow to build a linked list:

Suppose you arrive in Lausanne (Switzerland, lovely city! see Fig. 3.14) for a little sight-seeing trip. Because you are only staying a few hours, you don't bother reserving a hotel room, and instead you decide to leave your bags in a locker at the train station. So you find an empty locker, pay your fee, put your bags in the locker, and get your numbered key (let's say you got locker **#1342**).

Figure 3.14: A view of Lausanne, Switzerland. Looking southward across the lake is France. *Image: Switzerland Tourism, CC BY-NC 2.0 DEED.*

You go our and start exploring the city. It's a very interesting city and you buy a few things to bring home. First, you buy some Swiss chocolate, and to avoid it melting while you walk around you decide to go back to the train station and leave it there in another locker. You find an empty one, pay your fee, put the chocolates in there and take your numbered key (**#0789**).

Next you find some interesting pocket watches, buy one, and in order not to carry it around you head back to the station and put it in its own locker (same process as before), and take the numbered key (**#3519**).

The process repeats with you acquiring some books (left in locker **#6134**), a new digital camera (you left the old one in locker **#2156**), some more chocolate! (**locker #0178**), and a few t-shirts (**locker #9781**).

At this point, you notice that **you're walking around with a bunch of keys making noise in your pockets**. It's not fun. So you you start to wonder: **How could I store all my stuff (it doesn't fit in fewer lockers) in such a way that I need to carry only one key at any time, and yet I can still go and fetch any of my items whenever I want?**

After thinking about it for a while, you come up with this scheme:

Write down a list of all the lockers you have (in the order you got them): **#1342**, **#0789**, **#3519**, **#6134**, **#2156**, **#0178**, **#9781**.

Then you do the following (starting at the next-to-last locker and working backwards):

- Go to locker **#0178** and put it inside the key for locker **#9781** (together with the chocolates stored there, the key is small so it fits).
- Go next to locker **#2156** and store there the key for locker **#0178** (along with your old digital camera).
- Head to locker **#6134** and leave there the key for locker **#2156** (together with the books).
- Walk to locker **#3519** and put there the key for locker **#6134** (sharing the locker with the pocket watch).
- Move to locker **#0789** and leave there the key for locker **#3519** (together with the chocolates you bought first).
- Go to the first locker you got, **#1342** and store there the key for locker **#0789** (next to your luggage).
- Now you can walk outside again, carrying only the key for locker **#1342**.

You have just created an arrangement of lockers in which **you only have the key to the first one, and inside each locker you can find the key for the next one**. This is a **linked list**. **In this example, the links are the keys that open the next locker in the collection.**

Definition 3.1

The first locker in the list, the one for which we carry the key is called the **head of the list**. *The last locker, the one with no key inside is called the* **tail of the linked list**. ♣

If we draw a map of the items as they are stored in the locker room, we would expect it to look like Fig. 3.15.

Important things to note in the map in Fig. 3.15:

- Lockers are ordered (but not in increasing order of locker number!). The order is given by when they were added to your collection, and whatever locker happened to be available when you added each item.
- Each locker except for the last one **has a unique successor** whose numbered key is part of the locker's contents.
- The last locker (the tail of the list) has no key stored in it.
- The **key to the first locker** (the head of the list) is **not stored in any locker**, it's kept by you.

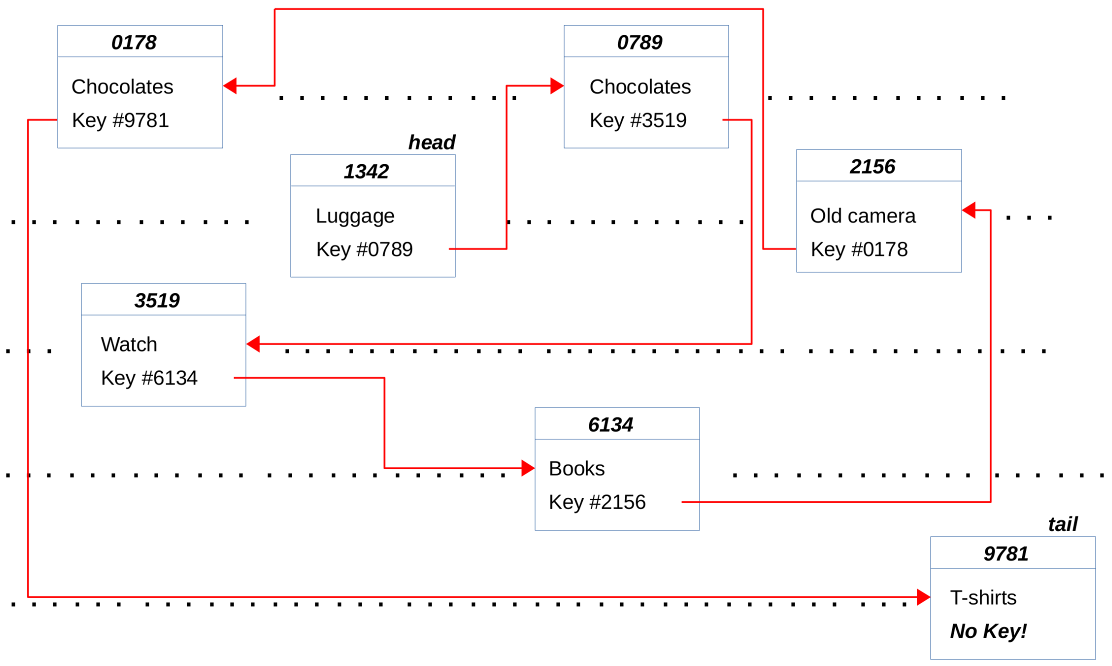


Figure 3.15: Map of the locker room at the train station after organizing all the items stored there into a **linked list**.

Questions:

- Would you ever expect the lockers to be ordered by increasing value of locker number?
- Which locker is the successor of locker #6134?
- Is the order of the lockers meaningful (does it provide any information about what's stored in the locker)?

3.5.1 Looking for something?

Suppose now that you have been walking for a while snapping pictures. Your new camera runs out of battery, but luckily you remember you bought a spare one and left it in the locker that contains the old camera.

Question: What is the sequence of actions you have to take to retrieve the spare battery from the locker with the old camera?

Because of the structure of a linked list, whenever we are looking for a specific item **we need to traverse the list**, starting at the head, and using the key in in each locker to open the next one in the list until we find the item that we want.

In this case, we would have to carry out the following actions:

- Use your key to locker **#1342**, look inside. This is not the locker you need, so use the key stored there to open the next locker **#0789** (don't forget to put the key back before closing **#1342**).
- Look in locker **#0789**. Chocolates! But we need a battery, so use the key stored there to open the next locker, **#3519**.
- Look in locker **#3519**, the watch is not what we're looking for, so use the key stored there to open the next locker, **#6134**.

- Look in locker **#6134**, it's books! Not what we are looking for. So take the key there and use it to open locker **#2156**.
- Look inside locker **#2156**. It's the old camera! Bingo! Fetch the spare battery, close the locker, and head out.

As you can easily see, that took some time and work. We will return to this later on.

Note

Remember: Whenever you we are using a linked list to keep a collection of items, searching for a specific item will require **traversing the list until we find it**. Unlike arrays, we can not simply go to any arbitrary item in the list – **we need the key**, and the key is stored in another locker. The only way to get to a particular item is to follow the links from one locker to the next until we arrive at the one that contains what we're looking for (or we reach the end of the list).

✍ **Exercise 3.4** Turns out all that walking has left you a bit sweaty, so you decide to change your shirt. **Write down the sequence of actions** that would be required for you to fetch a clean t-shirt from the ones you stored at the train station.

3.5.2 What if we need to store more things?

Suppose that you find a nice painting of a Swiss landscape that you want to bring home. You buy it, and you bring it back to the station.

Question: How can we insert (add) another locker to our collection?

There are several ways in which we can add new items to our collection. Whichever one we choose, we must carry out at the very least these three steps:

- Get a **new locker** to store things in, we will **get the key to this locker**.
- Put whatever we need to store in the newly acquired locker.
- **Link** the new locker to the rest of our collection. This is the crucial step for making sure our linked list works as intended as we add more items to it.
- **How to link the new locker to the list** depends on where in the list we want to insert it, and will affect how much work we need to do to add each item to the list.

Example 3.9 Let's store our newly bought painting in a locker, and **insert the new locker in our collection at the head of the linked list**:

- Reserve a new locker (we got **#4451**).
- Put the painting in the locker (we're lucky, it just fits!).
- **Link the new locker to the existing linked list at the head**. This means **the new locker will become the first locker** in our list, in effect, it will become **the head**. Locker **#1342** which was previously the head will become the second locker in the list.

- So we take the key we are currently carrying for locker **#1342** (our original **head of the list**) in the new locker. Locker **#4451** is now the head of the list so we take the key for it with us. Done!

After we completed the process above, our map for the locker room will look as shown in Fig. 3.16. The **new link** joining the locker we just acquired for our painting to the rest of the list is shown in green.

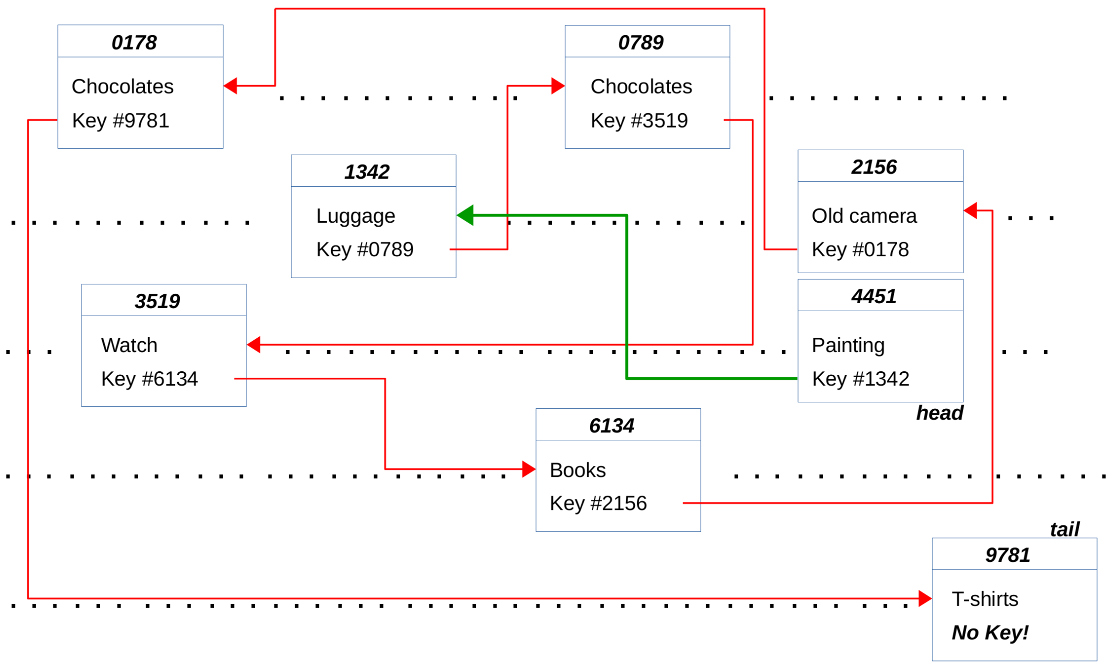


Figure 3.16: Map of the locker room after we add an item (a painting) to our collection **at the head** of the list.

The same process would allow us to add any number of items at the head of the list as long as there are unused lockers at the train station. The list will grow from the front end.

✍ **Exercise 3.5** Starting with an **empty list**, show a diagram of what the linked list looks like after we insert chocolates (locker **#2215**), Swiss cheese (locker **#0117**), a coo-coo clock (locker **#4152**), and a bunch of postcards (locker **#1890**), in that order, by **inserting each item at the head of the list**.

3.5.3 Inserting a new item at the tail of the list

Inserting new items **at the head** of the list is the most straightforward (least effort) way to insert a new item into the list. However it is not the only option. We can, with a bit more work, insert a new item **at the tail of the list**, so the list grows from the tail-end.

Example 3.10 Suppose we wanted to add the new locker with our painting (**#4451**) at **the tail of the list** (instead of at the head of the list). We would have to:

- Get the new locker **#4451** and its key.
- Store the painting in the locker, note that this locker will not contain a key since it will go to the end of the list.

- **Traverse the linked list** until we reach the **current tail** (easily recognized because it has no key in it). In this example, that would be locker **#9781**.
- Put the key to the new locker **#4451** inside the current tail of the list (**#9781**). This means locker **#9781** is **no longer the tail of the list** as it now has a key to another locker. At the same time this **links** the new locker to the list, **at the tail**.

Carrying out the process described above results in the map for the locker room that is shown in Fig. 3.17. Once again, the new link is shown in green.

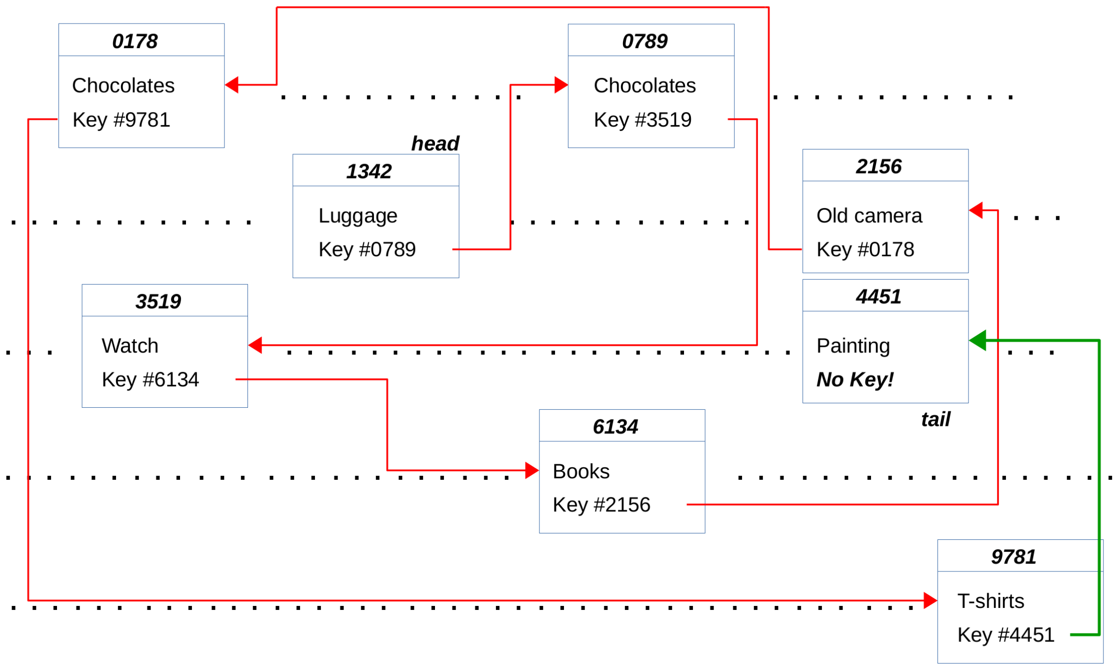


Figure 3.17: Map of the locker room after we add an item (a painting) to our collection **at the tail** of the list.

.

Note

Don't forget: Adding an item **at the tail of the list** involves **traversing the entire list**. This can be a lot of work! So why would we ever want to do this? Think about this little problem for a bit, and we will soon find applications for which adding items at the end of a list makes perfect sense.

✍ **Exercise 3.6** Starting with an **empty list** show what the linked list would look like if you carried out the following operations (the lockers you get are indicated for each item):

- Insert chocolates (**#0008**) at the **head of the list** (is this the same as inserting chocolates **at the tail** at this point?)
- Insert a bag with croissants (**#9501**) **at the tail** of the list
- Insert a bag of books (**#0546**) **at the head** of the list
- Insert a pair of t-shirts (**#6121**) **at the head** of the list
- Insert a pair of shoes (**#2222**) **at the tail** of the list

Questions: After the above operations are performed,

- What is the head of the list?
- What is the tail of the list?

3.5.4 Inserting at a location in-between existing items

The last option for inserting new items involves placing them **somewhere in-between existing things** in our list. This is the most involved operation (though as we will see every step makes sense if you think about how the lockers need to be organized). Like inserting at the tail, this is a type of insertion that makes sense for particular applications. Let's see how it's done.

Example 3.11 Suppose we wanted to store the painting right after the books (or, what amounts to the same thing, right before the old camera). The process would look like this:

- Acquire a new locker for the painting (**#4451**).
- Store the painting in that locker.
- **Traverse the linked list** until we **find the locker that contains the books** (**#6134**). At this point, we need to make sure the lockers end up in this order: **#6134** (books) $\rightarrow$ **#4451** (painting) $\rightarrow$ **#2156** (old camera).
- We take the key for locker **#2156** (old camera) from locker **#6134** (books).
- We store the key for locker **#4451** (painting) in locker **#6134** (books) - this links the painting to the list just after the books.
- We store the key for locker **#2156** (old camera) in locker **#4451** (painting - this links the old camera to the list just after the painting).

That's it. Notice that we don't have to do anything with the contents of locker **#2156**, as far as that locker is concerned, nothing happened! The resulting map of the locker room is shown in Fig. 3.18. The **updated links** are shown in green.

The only part of the process where we have to be really careful is when we're moving the keys around. However, if you take a moment to really understand why the steps above work, you'll be able to figure out the steps whenever needed, and you won't need to memorize anything. **You can always figure out the steps if you can draw what the lists should look like before and after adding the new item.**

✍ Exercise 3.7 List the steps needed to insert a bag of Swiss decaf coffee into our collection **in-between the chocolates and the t-shirts**. Make sure to list every step and clearly indicate **which keys end up in which lockers**.Show what the resulting list looks like after adding the coffee.

Ideas that you should be comfortable with at this point:

- How a linked list is organized
- How to search for a specific item in a linked list
- How to insert a new item at the head, tail, or in-between existing items

3.5.5 Searching for items in the list

Now that we know how a **linked list** works, how it is organized, and how to add items to it, the next operation we should figure out (as specified by the list of operations a **List ADT** has to support), is **search** - this means the

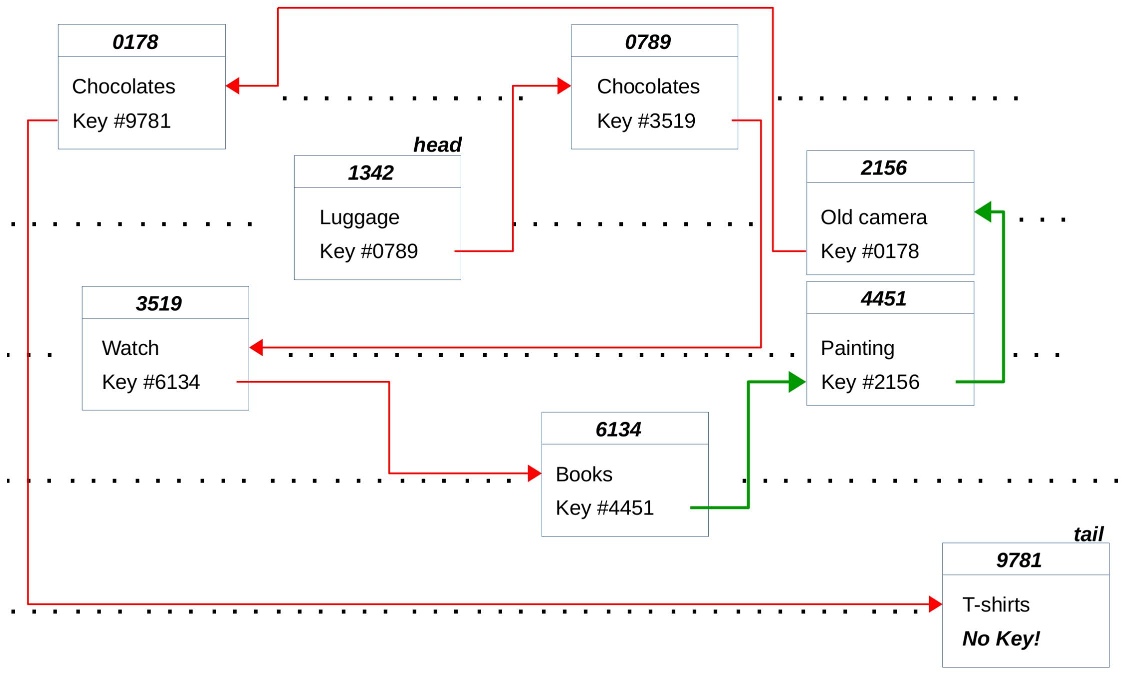


Figure 3.18: Map of the locker room after we add an item (a painting) to our collection **in between two other items** of the list.

process of finding a **specific item** in our collection.

As it turns out, we have already done that while we were figuring out how to **add items** either **at the tail** of the list, or **in between** specific items in the list.

Definition 3.2 (Searching)

In a **linked list**, **search** *is simply the process of* **traversing the list starting at the head** *and following each successive link in the list* **until we find the item we are looking for** *or* **we reach the end of the list**. ♣

We used **search** in order to get to **the end of the list** so we could add items **at the tail** of the list (we were searching for an item in the collection that occupied a locker where there was no key leading somewhere else). We also used **search** to find the item **after which we wanted to add a new item** to our collection.

In most applications of **linked lists** to real-world problems in computer science, we rely on **search** to obtain detailed information about the data items our collection is meant to organize and manage. For instance, in an application meant to store patient's medical records, we would often require to **pull up someone's complete record**, and this requires us to **use search to locate the entry in our linked list that contains that record**, so we can access the information stored therein.

3.5.6 Deleting items from the linked list

All the work of walking around acquiring things and storing them in lockers in a well organized linked list has made you very hungry. You decide to eat all the chocolates in one of your lockers, you remember there's two of them, and you're very hungry indeed so you decide to eat the first ones you find in your collection.

You head back to the lockers, and traverse your linked list until you find chocolates:

- Start at locker **#1342** (luggage), get key for locker **#0789**
- Go to locker **#0789** (chocolates). Found them! Eat all the chocolates!

After you've eaten the chocolates the locker is empty, so you decide to return the key to the locker rental office so that someone else can use that locker, but first you have to make sure the remaining lockers are still a linked list!

The situation we have at this point is like this: **#1342** (luggage, key for **#0789**) $\rightarrow$ **#0789** (no items, key for **#3519**) $\rightarrow$ **#3519** (watch) ... (the rest of the list). If we remove **#0789**, we need to make sure that locker **#1342** becomes linked to **#3519** which is the locker immediately after the chocolates that were eaten.

So to remove an item from the list we

- Find the item **right before the item we are removing** (this is called the **predecessor** of the item we are deleting).
- Take **the key from the locker that has the item we are deleting** and **store it in the predecessor**- which links the items **right before** and **right after** the one we are removing from the list.

In the case above, we need to take the key to locker **#3519** from locker **#0789** which is being removed, and store it in locker **#1342**. This will result in the following situation: **#1342** (luggage, key for **#3519**) $\rightarrow$ **#3519** (watch) ... (rest of the list). This is shown in Fig. 3.19.

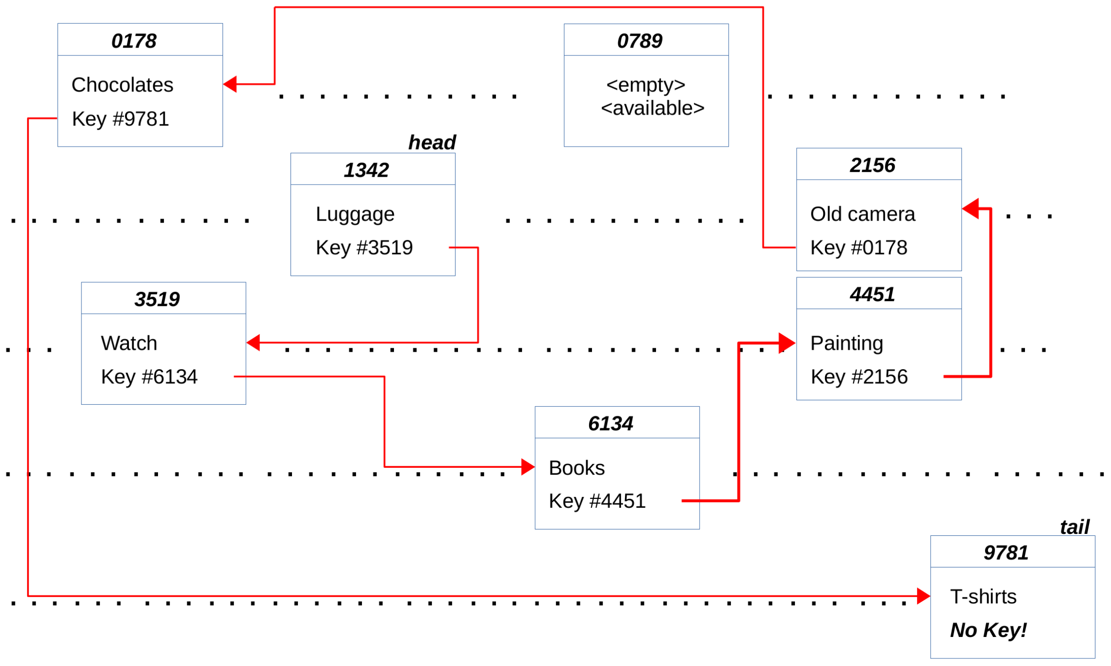


Figure 3.19: Map of the locker room after we remove the chocolates from our collection.

.

The locker **#0789** is no longer part of our list, and we can return the key to the rental office so the locker can be re used. Because our linked list is all about acquiring lockers on demand, and being able to acquire as many as we need to store our items, we should be good citizens and **never forget to return a locker we no longer need** so it can be re-used by others, or by ourselves at a later time.

Questions:

- Does the same process work if we are removing the item at the tail of the linked list?
- Does the same process work if we are removing the item at the head of the linked list?

You may be wondering why we have developed the above example of linked lists without any actual code. The reason is that the same process applies to linked lists independently of what language we are programming with, or what items we are storing there. So, understanding how the list works independently of code will allow you to implement a linked list in any language, for any application, and for storing any type of data. In effect we have defined a **linked list ADT**.

This is precisely the kind of conceptual understanding that is essential to acquire. Implementing the linked list will help refine and solidify your understanding, but do not forget: The **concept**, **process**, and **organization** of the linked list are more important than any specific implementation.

3.6 Implementing a Linked List in C

Up to this point, we have been discussing linked lists at a conceptual level, as an **Abstract Data Type** that can be implemented in any programming language, and in many different ways. It is now time for us to look at an actual implementation of the **linked list ADT**.

Definition 3.3 (A specific implementation of an ADT is called a data structure)

The difference is important: There may be many different ways to implement a particular ADT (even using the same programming language), and implementations of the same ADT in different languages may look completely different. The **data structure** *on the other hand is* **programming-language specific, and implementation dependent**. ♣

A properly designed **data structure** has to comply with the expectations described by the corresponding **ADT**, that means that it has to organize information in the way the **ADT** describes, and it has to support every operation the **ADT** specifies must be available for the information kept in the collection. In the case of a **linked list**, the **ADT** specifies that items have to be stored in sequence (the order doesn't matter), and the operations supported are **adding (inserting) items** into the collection, **searching (finding) specific items**, and **deleting items** from the collection.

What we are about to do is to create a **linked list data structure** in **C**. This involves the following steps:

- Setting up a new compound data type (**CDT**) to store one item in the list. Each individual item is usually called a **node** in the list.
- Setting up a **head pointer** to keep track of the **head of the list**.
- Writing a function to **create a new empty node on demand** - remember we must be able to add items when we want to, and only use memory for items that are actually in the list.

- Writing a function to **insert properly filled nodes** into the list - this means **nodes** that have all the information required by one item in the collection.
- Writing a function to **search** for a specific item.
- Writing a function to **delete (remove)** a specific item from the list.

It seems like a bit of work, but as we shall see the process is independent from the type of information that the linked list contains, so once you know how to do the steps above for items of one kind, you can do it for items of any other kind.

3.6.1 Creating a node CDT

The **general structure of a node** in a linked list is

```
----------------
|    DATA       |
|               |
| link to next --------->
----------------
```

The **node** itself is just a big box with 2 parts: **a data payload** that consists of all the information we need to store **for a single item in the collection**, and a **link to the next entry** in the list.

The **data payload** can be anything. From a simple data type such as **int** or **float**, to a chunky **compound data type** that contains multiple fields, each of which has its own data type (and each of which could itself be a **CDT**). The **data payload** can contain **pointers** to information stored elsewhere. For example, we could create a linked list to store information about all the files stored in a USB memory stick, each **node** could contain information about the file such as its name, the date it was created, the file size, the file type, and **a pointer** to the memory location in the USB stick where the actual data for the file is stored.

The point is that the **data payload** can contain anything we want. The structure of the list doesn't depend on what kind of data it stores. It just provides a means to keep it organized.

The **link to the next entry** in the linked list is just **a pointer** that stores the **memory location** (the locker number) of the box that contains the next item in the linked list. We have used pointers before to access information our program needs, the **next item pointer** works just like any other pointer.

We have to use pointers because as we have learned

- We **don't know where in memory a new node will be placed** - it depends on where there's space the moment we ask for a new node to be created.
- We will **request space for nodes on demand**, and will request as many as we need but no more - we can not use an array for this.
- In **C**, we need pointers to **allow functions to access/change variables declared outside their scope**. All the functions that work on the linked list will have to do this, so we need the pointers.

Let's see how we declare a **linked list node** for a simple data type.

Example 3.12 Declare a linked list node where **each node stores a single int value**.

```
        typedef struct int_list_node
        {
            int stored_integer;          // DATA
            struct int_list_node *next; // Link to next entry
        } int_node;
```

We have already seen that we use **typedef** to create new **CDTs**. A linked list node is a **CDT** and is defined in exactly the same way. The first line

```
        typedef struct int_list_node
```

tells the compiler that we are defining a new compound data type called **int_list_node** (a node for a linked list containing integers).

The next couple lines

```
            int stored_integer;           // DATA
            struct int_list_node *next;  // Link to next entry
```

Define the contents of this node: one **int** value called **stored_integer**, and a **pointer to the next node** in the linked list (which is of type **int_list_node**). In most linked lists, this pointer is called **next**. The last line

```
        } int_node;
```

tells the compiler we want to call our new data type **int_node**. Thereafter we can go ahead and declare variables for nodes in our linked list like so:

```
        int_node a;       // A variable of type int_node
        int_node *head;   // A pointer to an int_node
```

Let's see how we would use our new data type in a little program.

Example 3.13

```
    #include<stdio.h>
    #include<stdlib.h>

    typedef struct int_list_node
    {
        int stored_integer;           // DATA
        struct int_list_node *next;  // Link to next entry
    } int_node;

    int main()
    {
        int_node   a_node;
        int_node   *node_ptr=NULL;

        a_node.stored_integer=21;
        a_node.next=NULL;

        node_ptr=&a_node;
        node_ptr->stored_integer=17;

        printf("The value contained in the node is %d\n",node_ptr->stored_integer);

        return 0;
    }
```

Compiling and running the code above we get:

```
>/a.out
The value contained in the node is 17
```

Let's see what this does in memory to fully understand out little program.

First, **main()** declares two variables

```
    int_node   a_node;
    int_node   *node_ptr=NULL;
```

The first one is a linked list node called **a_node**, the second one is a pointer to a variable of type **int_node**. In memory, these lines will reserve one box of the right size to hold an **int_node**, and one box for a **pointer** to **int_node**; as well as space for **main()'s** return value. This is shown in Fig. 3.20.

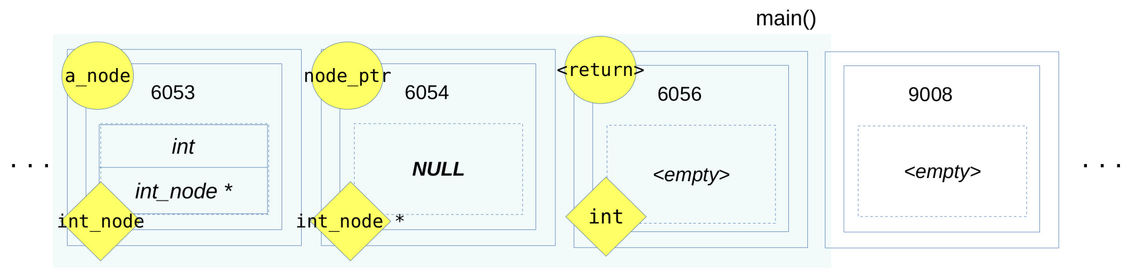


Figure 3.20: Memory model for the code in Example 3.13 just after space has been reserved for the program.
.

Note that:

- The box containing the list node has two parts: an **int**, and a pointer to an **int_node** so we can link this box into a list.
- The **node_ptr** on the other hand is just a pointer, it doesn't have two components despite being a pointer to a variable of type **int_node**. Initially it is set to **NULL** indicating it's not pointing to anything.

Next, the program fills-in the data in the variable **a_node**, there's two fields to fill, and they are set to appropriate values.

```
    a_node.stored_integer=21;
    a_node.next=NULL;
```

notice that the **next** pointer is being set to **NULL**. Whenever we create a **new node**, we must make sure that the **next** pointer is set to **NULL** and it does not receive a different value **until the node is linked to a list**. Not setting the **next** pointer to **NULL** is a common source of bugs in programs that work with linked lists. In the memory model, the situation is as shown in Fig. 3.21.

Next we get a pointer to the list node **a_node**, use it to change the value of the data in the node; and then we print out the node's data contents:

```
    node_ptr=&a_node;
    node_ptr->stored_integer=17;

    printf("The value contained in the node is %d\n",node_ptr->stored_integer);
```

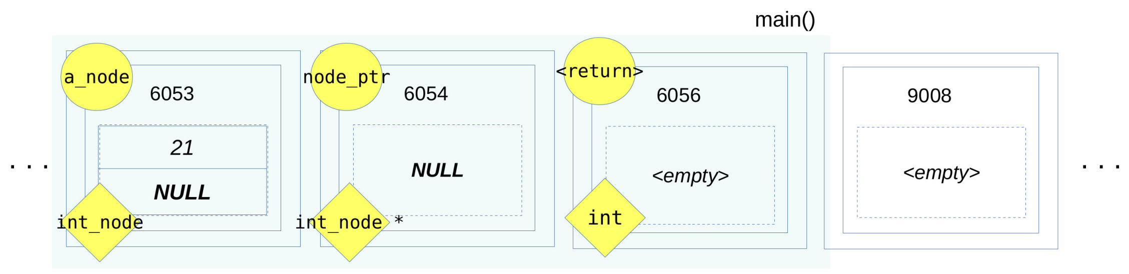


Figure 3.21: Memory model for the code in Example 3.13 after **a_node** has been filled with information.

.

The first line is read as **get the address of** *a_node* **and store it in** *node_ptr*, then we access the node's content using our pointer (remember, when we have a pointer to a compound data type, we can access its different fields using the **'->'** operator). In this case the line reads **make the value of the** *stored_integer* **at the node whose address is in** *node_pointer* **equal to 17**. The last line prints out the node's stored integer (using the pointer to access it). As expected, it prints out **17**. The memory model will look as shown in Fig. 3.22.

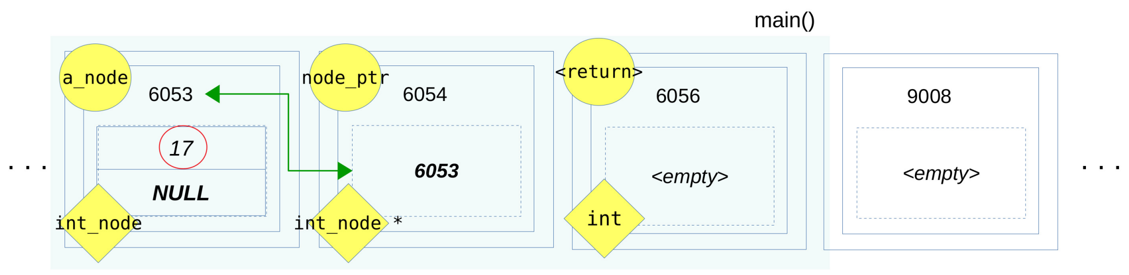


Figure 3.22: Memory model for the code in Example 3.13 just before the program ends.

.

You can see that **node_ptr** contains nothing more than the address for **a_node**. If we had a function that needs to access/modify the data in **a_node**, we could pass **node_ptr** to it.

The example above is to show you how we define a **node data type**, how we can declare variables and pointers to nodes, and how we can use them to access and modify data in the node. However, we started this section by saying **we want to be able to create nodes on-demand**, as new data items are added to a collection. We can not do this with variable declarations that are written into the code.

We will now see how to create nodes on-demand (**this is called dynamic memory allocation**, which is nothing other than a fancy term for getting space for data whenever we need it). We will see that **the only way to deal with such data is by using pointers**. We should be thinking of our original restaurant review app, so let's apply what we know to create a linked list of restaurant reviews, and see how we can generate new restaurant reviews on-demand to put in our list.

3.6.2 Declaring a linked list node for reviews

Remember our Review data type (we have already seen how it works and how to use it):

```
#define MAX_STRING_LENGTH 1024
```

```c
typedef struct Restaurant_Score
{
       char restaurant_name[MAX_STRING_LENGTH];
       char restaurant_address[MAX_STRING_LENGTH];
       int score;

} Review;
```

Let's now declare **a node for storing reviews** in a linked list. Just like before, our node will contain two parts: A **variable to hold one Review**, and a **pointer to the next node** in the linked list.

```c
typedef struct Review_List_Node
{
    Review rev;
    struct Review_List_Node *next;
} Review_Node;
```

This will create a new data type called **Review_Node** that contains one **Review**, and **a pointer** to the next entry in a linked list.

Take a moment to compare this node definition with the one for the **int_list_node** above, and you will see that the only change is that the data component of the node is now a variable of type **Review**. Other than that it works exactly the same way. This shows that **creating a node for a linked list works the same way for any data type**.

3.6.3 Creating nodes on-demand

Since we must be able to create nodes on-demand, we need to write a function that will

- **Reserve space** for a new **Review_Node**.
- **Initialize the contents** of the newly reserved node - to reasonable **default** values.
- Provide our program with **a pointer to the new node** so we can access/modify data inside it, and so we can link it to a list.

Here is how we would do that for review nodes, but note that the same process will apply to nodes containing any other data type:

```c
Review_Node *new_Review_Node(void)
{
  Review_Node *new_review=NULL;  // Declare a pointer to locker that contains a Review_Node

  new_review=(Review_Node *)calloc(1, sizeof(Review_Node)); // Reserves memory space!

  // Initialize the new node's content with reasonable default values that show
  // it has not been filled with actual data. In our case, we set the score to -1,
  // and both the address and restaurant name to empty strings ""
  // Very importantly! Set the 'next' pointer to NULL

  new_review->rev.score=-1;
  strcpy(new_review->rev.restaurant_name,"");
  strcpy(new_review->rev.restaurant_address,"");
  new_review->next=NULL;
  return new_review;              // This returns a *pointer*, NOT a Review_Node.
}
```

Let's look at the code above in detail. You will use very similar code for creating nodes on-demand for any linked list you will write in **C** (as well as for many other data structures we will study later on). First, the function declaration:

```
Review_Node *new_Review_Node(void)
```

It states that the function called **new_Review_Node** has **no input arguments**, and **returns a pointer** to a **Review_Node**. Inside the function's body, we have one variable declaration:

```
  Review_Node *new_review=NULL;       // Pointer to the new node
```

This is just a **pointer** to a **Review_Node** and it's initially set to **NULL** to indicate it's **UN-assigned**. The actual work of allocating a new **Review_Node** is done here:

```
  new_review=(Review_Node *)calloc(1, sizeof(Review_Node));
```

The syntax here requires a bit of care to understand. The function **calloc()** is a library function that is used to **reserve memory on-demand**. It takes in two parameters:

```
  calloc( # of items , size of each item in bytes)
```

In the case above, we are requesting **one item** whose size is the **size of a Review_Node**. Luckily for us we have a helpful **sizeof()** operator that returns the size in bytes of any data type known to the compiler - which includes any **CDTs** we have previously declared.

What does calloc() do?

- It **finds an available place** in memory that has the requested capacity
- It **reserves** that memory for use by our program
- It wipes-out the contents of that memory space with zeros - so it will **not** contain **junk**
- It returns a pointer to our reserved chunk of memory

The function **calloc()** returns a pointer without any attached data type (it's a simple function, it doesn't know what we want to do with the memory), so the line

```
  new_review=(Review_Node *)calloc(1, sizeof(Review_Node));
```

takes the returned pointer, **type-casts** it to a **pointer for a variable of type Review_Node**, and stores it in our pointer variable **'new_review'**. That's a lot to take in! So let's review it slowly in steps:

- We declared a pointer **new_review** to a **Review_Node** box, which we expect to request and reserve on-demand.
- We then used **calloc()** to reserve memory space for the new node. It gives us a pointer to a clean box suitable to store a **Review_Node**.
- We stored that pointer so we can use it to access/modify the information stored in the **Review_Node** box.

We will see how all of this works in memory in a moment. Let's just finish going through the **new_Review_Node()** function. The last part of this function initializes (fills-in) the values of our newly acquired **Review_Node** with **default values** that show the node has not been updated with actual data.

Note

This is an important step and helps us avoid bugs caused by trying to use information in nodes that have been **created** but still **contain no valid information**.

In the case above, the code sets the **'score'** to **-1**, and initializes the restaurant's name and address to empty strings (""). It then sets the **'next'** pointer to **NULL**. This is an essential step as it ensures that if the **'next'** pointer has **any value other than NULL**, then the node must be part of a linked list. **Always initialize pointers in newly created nodes to NULL**.

To fully understand what the function above does, let's see what happens in memory if we run a little program that creates a single **Review_Node**, fills the new node with information, and prints that information out.

Example 3.14 The program below **dynamically allocates** a **Review_Node**, fills the **Review** within that node with information, and then prints the information inside the **Review**. Pay close attention to how **calloc()** is used to reserve memory on demand, how **a pointer** us used to access the newly reserved box, and how the program can access **data fields** that are inside a double wrapping: They are stored inside a **Review CDT**, which then is packed inside a **Review_Node CDT**.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define MAX_STRING_LENGTH 1024

typedef struct Restaurant_Score
{
    char restaurant_name[MAX_STRING_LENGTH];
    char restaurant_address[MAX_STRING_LENGTH];
    int score;
} Review;

typedef struct Review_List_Node
{
    Review rev;
    struct Review_List_Node *next;
} Review_Node;

Review_Node *new_Review_Node(void)
{
    // This function creates a new box for a 'Review_Node',
    // initializes the information stored in the Review_Node,
    // and returns a pointer to the new node.

    Review_Node *new_node=NULL;  // Pointer to the new node

    new_node=(Review_Node *)calloc(1, sizeof(Review_Node));
    if (new_node==NULL)
    {
        // *ALWAYS* after using calloc, check if we actually were able
        // to reserve the memory we wanted. calloc() returns NULL if it
        // is not able to reserve the requested memory, we must check for
        // that and handle the situation in a way that is appropriate
        // to the program we are writing. In this case, print an
        // error message and return NULL (so the function that called
        // new_Review_Node() knows it did NOT get a node and can
```

```
            // also react accordingly).

            printf("new_Review_Node(): Error - there is no memory available, unable to
                reserve space!\n");
            return NULL;
        }

        // Initialize the new node's content with values that show
        // it has not been filled. In our case, we set the score to -1,
        // and both the address and restaurant name to empty strings ""
        // Very importantly! Set the 'next' pointer to NULL

        new_node->rev.score=-1;
        // The line above is important. We have a pointer to a Review_Node
        // box (new_node), inside that box is a Review *variable* called
        // 'rev', and inside 'rev' there is a field called 'score' that we
        // want to update. So the instruction states:
        // Use the pointer 'new_node' to access the 'rev' variable (using
        // the '->' operator, since 'new_node' is a pointer. Once we have
        // access to 'rev', access the 'score' field (because 'rev' is a
        // regular variable, use the '.' operator) and set it to -1
        // Similarly, the instructions below will update the name and address.

        strcpy(new_node->rev.restaurant_name,"");
        strcpy(new_node->rev.restaurant_address,"");
        new_node->next=NULL;

        return new_node;
    }

    int main()
    {
        Review_Node *my_node=NULL;

        my_node=new_Review_Node();
        if (my_node==NULL)
        {
            // This would happen if new_Review_Node() didn't get memory,
            // so we can't continue.
            printf("Could not reserve memory for a new Review_Node, ending the program now.\
                n");
            return 1;     // Return non-zero to indicate that an error occurred.
        }

        strcpy(my_node->rev.restaurant_name,"Veggie Goodness");
        strcpy(my_node->rev.restaurant_address,"The Toronto Zoo, Section C");
        my_node->rev.score=3;

        printf("The review node contains:\n");
        printf("Name=%s\n",my_node->rev.restaurant_name);
        printf("Address=%s\n",my_node->rev.restaurant_address);
        printf("Score=%d\n",my_node->rev.score);
        printf("Link=%p\n",my_node->next);

        free(my_node);

        return 0;         // Return 0 because everything went as expected.
    }
```

Compiling and running the code above produces:

```
>./a.out
The review node contains:
Name=Veggie Goodness
Address=The Toronto Zoo, Section C
Score=3
Link=(nil)
```

Let's see exactly what is happening when we run the code above. First, **main()** declares a pointer variable to a **Review_Node**. This means whatever memory address is stored here, we can expect at that location to find a box containing a **Review_Node**. The pointer is initialized to **NULL** as should be done with any pointers we declare. The initial situation in memory looks like that shown in Fig. 3.23.

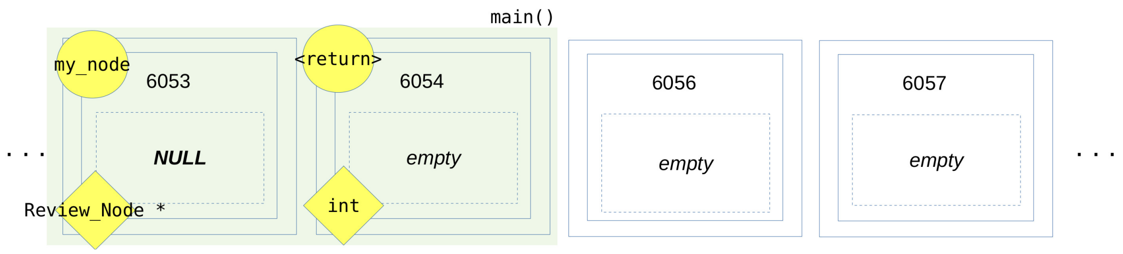


Figure 3.23: Memory model for the code in Example 3.14 just after space is reserved for **main()'s** variables.

Things to note

- The pointer **my_node** is the only variable declared in **main()**
- It is **not** a **Review_Node**, all that it can store is a **memory address**
- It is initialized to **NULL** to indicate it is unassigned at the moment
- Let's not forget about **main()'s** return value!

Next we have a call to **new_Review_Node()**,

```
my_node=new_Review_Node();
```

the function **new_Review_Node()** declares a **single pointer** variable to a **Review_Node**, and also has a **return value** that is a **pointer to a Review_Node**. These need to be reserved in memory as shown in Fig. 3.24.

Note that neither **main()** nor **new_Review_Node()** have declared any actual **variables** of type **Review_Node**. We are using **pointers** exclusively. Next, the **new_Review_Node()** function **dynamically allocates** memory for the new **Review_Node**, and initializes it to reasonable **default values**. The result of this is shown in Fig. 3.25.

Things to note:

- The newly reserved **Review_Node** is shown outside **any of the functions** in the program. **It does not belong to either main() or new_Review_Node()**.
- It **doesn't have a name tag** because **it is not a local variable** declared by a function.
- Since it doesn't have a name tag, **the only way to get to it** is by having its address (**#9871**) in a **pointer**. The pointer **new_node** stores the address of the newly created node.
- The new **Review_Node** has two fields as expected: one field of type **Review** that we called **'rev'**, and **a pointer** called **'next'** that can be used to link this node to a linked list.

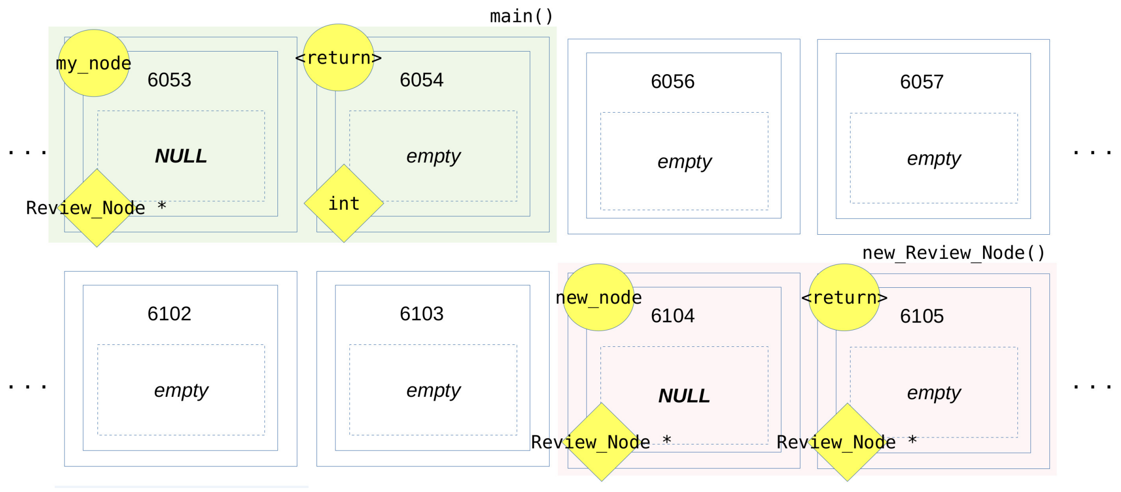


Figure 3.24: Memory model for the code in Example 3.14 just after the call to **new_Review_Node()** has reserved space for the function's variables.

.

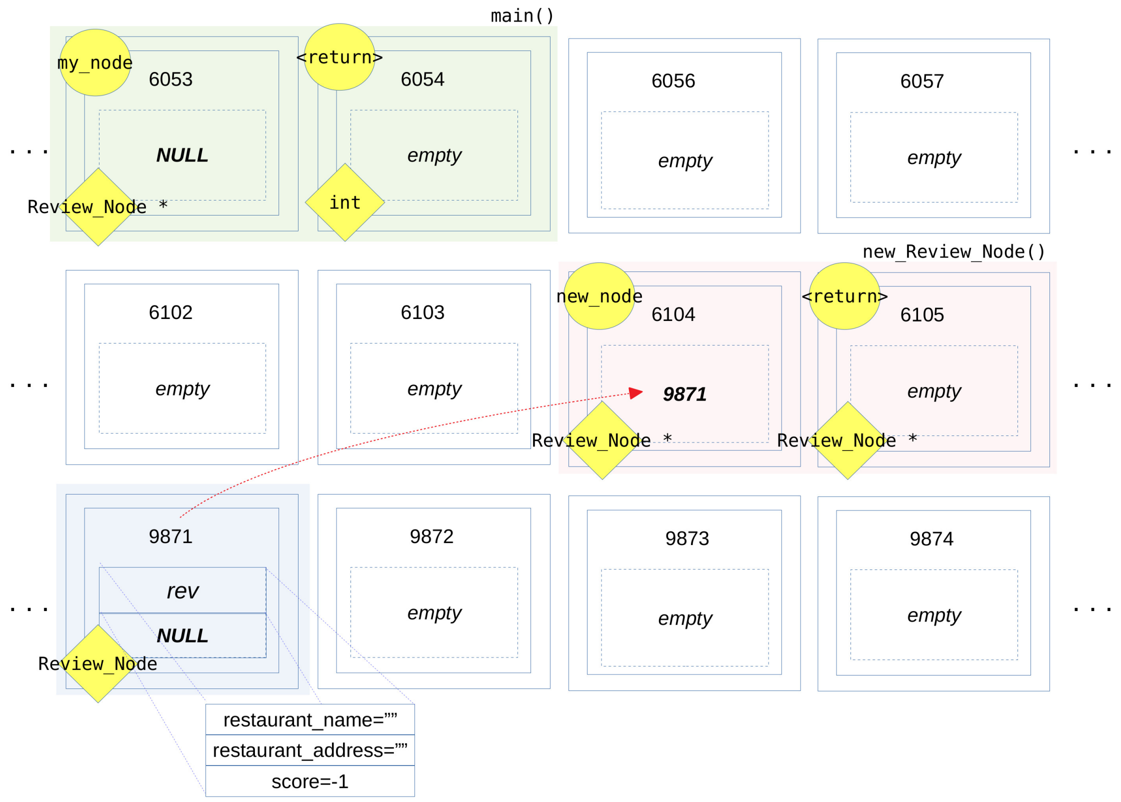


Figure 3.25: Memory model for the code in Example 3.14 after the function **new_Review_Node()** has reserved memory for a new **Review_Node** and filled it with **default** values.

.

Finally, the **new_Review_Node()** function **returns the pointer** to the newly allocated node back to **main()**, it gets stored in the **'my_node'** variable, giving **main()** access to the box for the new node, as shown in Fig. 3.26.

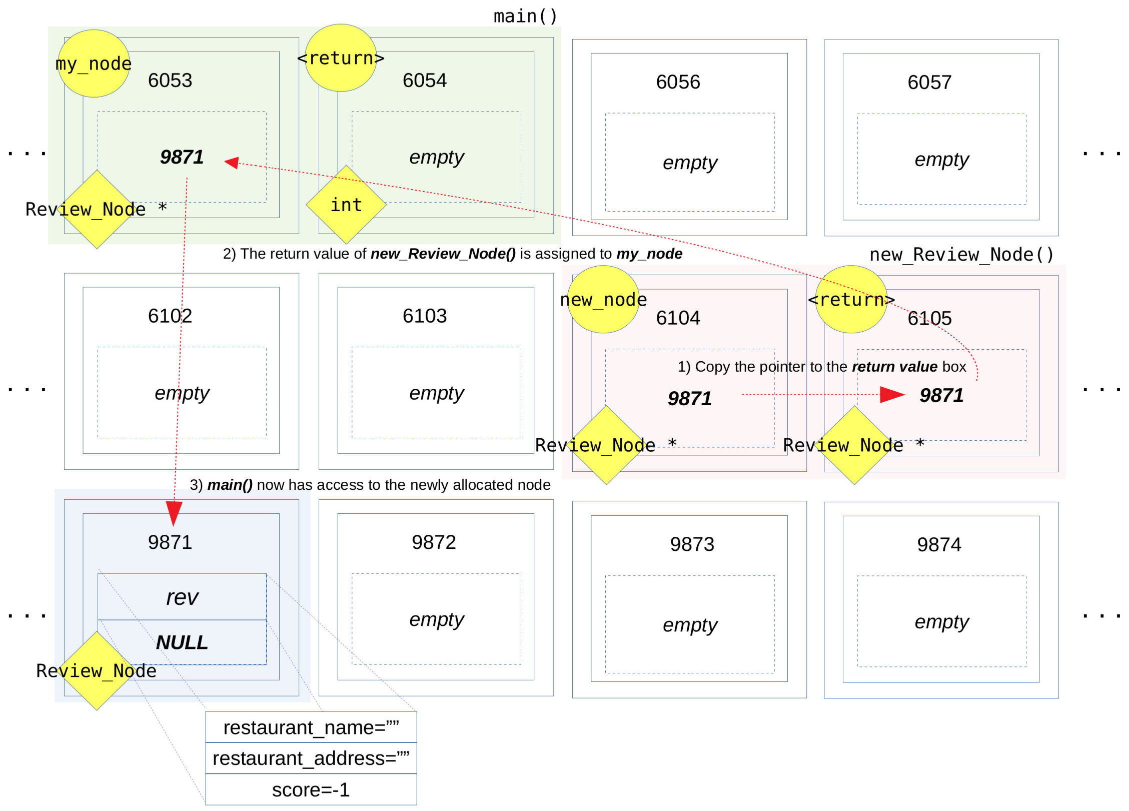


Figure 3.26: Memory model for the code in Example 3.14 once the **return statement** copies the pointer to the newly allocated node back to **main()'s** pointer variable **'my_node'**.

.

Its work completed, memory reserved for the function **new_Review_Node()** is released. Importantly **the box for the newly allocated node is not released** because it does not belong to the function, it is **not attached** to any function in the program and is not released when functions end. We will get back to this point in a moment.

At that point, **main()** has access to the newly allocated box, and can use its pointer **'my_node'** to fill-in the new node with information. The situation is shown in the memory model in Fig. 3.27.

The syntax for accessing information stored within a node is worth careful thought:

```
      my_node->rev.score=3;
//                      ^---- We want to update the score for this review
//                  ^-------- 'score' is a field in 'rev', which is a variable
//                             so we use the '.' operator to access it
//             ^------------- 'rev' is a field in the Review_Node box that
//                             is stored at the address in 'my_node', so we
//                             use the '->' operator to access it.
//     ^--------------------- 'my_node' is a pointer to a Review_Node
//                             and was declared in main()
```

a similar process is used to access the other fields in **rev**.

The last step in the program, right before the program ends, is **releasing the memory we have dynamically allocated**. This is essential whenever we use **dynamic memory** to store information. The reason for this is that, as

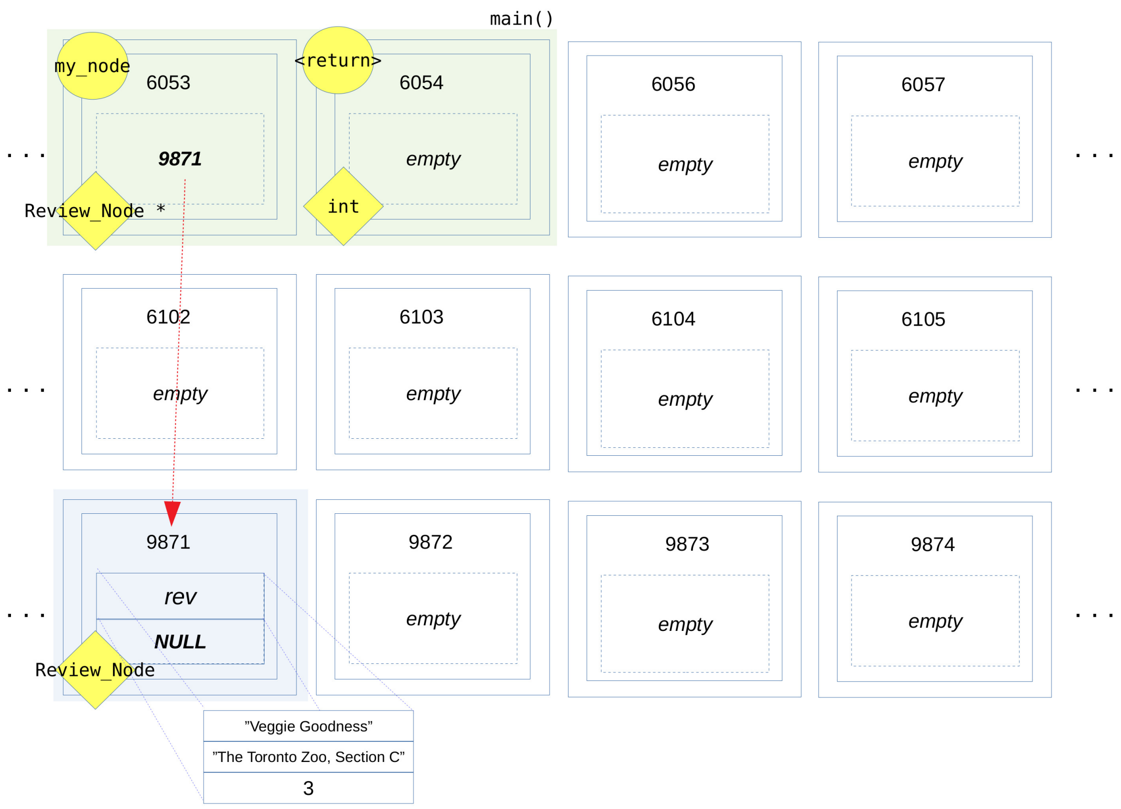


Figure 3.27: Memory model for the code in Example 3.14 after **main()** has filled-in the new node with information.

.

shown in the memory model, any boxes that our program reserves dynamically are stored **outside functions**, and have **no attached tags**. They are **not local variables** and **are not released automatically** when the function that reserved them ends. We have to clean up after ourselves.

Note

The process for releasing memory that has been dynamically allocated by our program uses a built-in function called **free()** that takes a single input argument: **a pointer to a chunk of memory that was dynamically reserved by our program**. The function releases the memory for use by other programs (or by our own program at a different point in time).

The last line in our program right before the final return statement releases memory for our newly acquired node

```
    free(my_node);
```

the process is straightforward but we have to be careful:

- For each data item we dynamically allocated, there has to be a corresponding **free()**.
- Any **dynamic memory** we acquired that we do not release with **free()** becomes a **memory leak**.
- We **can not free()** a chunk of memory more than once (if we try, the program will be terminated) - it is a bug.
- We **can not free()** a **NULL** pointer (if we try, the program will be terminated) - this is also a bug.
- We **can not free()** local variables, input arguments, or a return value - only dynamic memory can be freed in this way.

In summary: From the above example, we have seen how to build a **CDT for a node that contains a Review**, how to implement **a function that dynamically allocates a new node when we need it**, and that **returns a pointer** to the new node (properly initialized with **reasonable default values**); and how to use **this pointer** to access and update information stored in our dynamically allocated node. We will use all of these ideas very shortly in order to implement a fully-working linked list of restaurant reviews.

Note

Why did we bother with so much detail? In a different course, the code we wrote may have been explained in a much more compact way. In particular, the line

```
my_node=new_Review_Node();
```

could just have been explained by saying *the function* **new_Review_Node()** *allocates a new* **Review_Node**, *and returns its address*. This is an accurate statement, but it doesn't help us really understand what is going on when we reserve memory on-demand, or the process we have to follow if we ever have to (and we will have to!) write code that creates and initializes different types of data items. So, **it's worth going through the entire process once**, in great detail, making sure we understand every single step.

At this point, given all the examples we have done of how variables, pointers, compound types, and function calls are processed, we should have a pretty good understanding of what happens when we perform a sequence of operations in **C**. So, from now on, we will spend less time looking at the very low-level detail of how things change in memory when we run code, and **start looking at programs at a higher level, focusing on the conceptual aspects** of what we're doing - this is unavoidable since we will be implementing more complex programs and looking at every single step in them would take too long and wouldn't teach us anything new.

But we should not forget - **C** is a very simple and straightforward language that doesn't do anything we didn't ask it to do. **We can always figure out exactly what is going on** if we **think in terms of boxes in memory** that correspond to the data our program is working with, **and operations on these boxes**. Whenever we are not sure about what is happening, we should take a blank sheet of paper and a pencil, and draw a memory diagram, then make sure we understand what our code is doing step by step.

✍ **Exercise 3.8** Write a little program that

- Creates an **int_node CDT** which represents a node in a linked list where the data items are single integer values.
- Has a **function to allocate and initialize** a new **int_node**.
- Creates a new **int_node** in **main()**, and **uses a pointer** to update its integer value to 42.
- **Uses the pointer** to print out the contents of the **int_node**.
- **Releases** the memory for the **int_node** before the program ends.

3.7 Building a linked list of reviews

At this point we have all we need to create a linked list of restaurant reviews:

- We know how to **define a CDT** to store the data for a single review.
- We know how to **define a linked list node CDT** that we can use to link reviews into a list.
- We know how to write a function that **allocates new review nodes on-demand** and returns a pointer to the newly created nodes so we can access/modify information within.
- We know how to **read user input from the terminal** so we can obtain information to fill our reviews.

It's time we put everything together into a little program that is able to read review information from the terminal, fill-in the information typed in by the user into review nodes allocated on-demand, and link these nodes to form a linked list.

In order to complete this program we will need to:

- Implement a function to **initialize** a new (empty) linked list.
- Implement a function to **insert** a newly created review into the linked list.
- Implement a function to **search for** specific reviews we wish to inspect.
- Implement a function to **print the reviews** in the list whenever we want.
- Implement a function to **delete** a specific review the user no longer needs.
- Add code to **main()** that allows the user to choose what they want to do, and obtains any review information that is needed.

We will be looking at this code at a higher, more conceptual level, and stop only to look at details when such details illustrate a new idea we haven't seen before.

✍ **Exercise 3.9** Write in pseudocode the steps we need to implement as per the description below (we will start with only some of the linked list operations we need to implement, and extend our implementation afterwards).

- Gives the user a choice between: a) Entering a new review, b) Printing out all the reviews entered thus far, or c) Exiting the program.
- If the user chooses **a)** the program should carry out all the steps needed to add the new review to the linked list of reviews.
- If the user chooses **b)** the program will traverse the list printing out each review in turn.
- If the user chooses **c)** the program releases all memory allocated to the linked list, and exits.

Having completed the exercise above, have a look at how we would implement these steps in main().

```
int main()
{
  Review_Node *head=NULL;
  Review_Node *one_review=NULL;
  char name[MAX_STRING_LENGTH];
  char address[MAX_STRING_LENGTH];
  int score;
  int choice=1;

  while (choice!=3)
  {
    printf("Please choose one of the following:\n");
    printf("1 - Add a new review\n");
    printf("2 - Print existing reviews\n");
    printf("3 - Exit this program\n");

    scanf("%d",&choice);
    getchar();

    if (choice==1)
    {
        // Here we need code to add a new review to the linked list
    }
    else if (choice==2)
    {
        // Here we will add code to print the existing reviews
    }
  }

  // User chose #3 - Release memory and exit the program.
}
```

The code above is not complete (and is also missing the **#include** statements, and the declarations for the various **CDTs** we need). But, importantly, it contains the different sections we need to complete in **main()** in order to implement all the functionality requested. It already does two important things:

Firstly, It declares a new, empty linked list:

```
  Review_Node *head=NULL;
```

As you see, it's just a pointer to a **Review_Node**, initially set to **NULL**. This is how we create any empty linked list in **C**.

Question: How do we check if a linked list is empty?

Secondly, it provides a loop that **prompts the user** to choose a number from 1 to 3. Depending on the user's choice, different sections are executed. If the user chooses **3**, the loop exits.

Question: What does the loop do if the user inputs anything other than values in 1-3?

Note

When you are writing a complicated program, it is a good idea to write a little **driver function** (in the case above, we used **main()** for this but it could be a separate function called by **main()**) that has a loop like the one above, and allows you to **test different parts** of the program separately giving you control over which part gets tested, in what order to test the different program components, and what information to pass to each of them. We will get back to this topic at the end of the Chapter.

Let's now start filling in the missing parts of the implementation. First, let's have a look at the code for option **1**, it should **insert a new review** into our linked list.

3.7.1 Inserting a new node into the linked list, at the head

```
  if (choice==1)
  {
     // Request a box for a new review node

     one_review=new_Review_Node();     // We saw how this works in the previous section!
     if (one_review==NULL)             // Remember to check that we actually got a node..
```

```
    {
          printf("main(): Error - there is no more memory, unable to create a new linked list
              node!\n");
          exit(1);                     // This ends the program at this point
    }

    // Read information from the terminal to fill-in this review
    printf("Please enter the restaurant's name\n");
    fgets(name, MAX_STRING_LENGTH, stdin);

    printf("Please enter the restaurant's address\n");
    fgets(address, MAX_STRING_LENGTH, stdin);

    printf("Please enter the restaurant's score\n");
    scanf("%d",&score);
    getchar();

    // Fill-in the data in the new review node
    strcpy(one_review->rev.restaurant_name,name);
    strcpy(one_review->rev.restaurant_address,address);
    one_review->rev.score=score;

    // Insert the new review into the linked list
    head=insert_at_head(head,one_review);
 }
```

The code above uses the function we wrote before, **new_Review_Node()** to allocate and initialize a new **Review_Node**. You already know how this function works, and what happens in memory when we call it. In the code above, we can simply assume that we obtain a pointer to a newly allocated **Review_Node**. But notice we always check to see if a problem occurred (in which case **new_Review_Node()** will return **NULL**).

The next step is to obtain information from the user to fill-in the review. Once we have this data, we can update the fields inside the **'rev'** variable that is itself a field of the **Review_Node**.

The final step is to **insert the new node into the linked list**. For this we have a function (not yet implemented!) called **insert_at_head()**. Remember we discussed above the three different ways in which we can insert a node into a list: **at the head**, **at the tail**, or **in between** existing nodes.

Here **we will insert new nodes at the head** because our program **does not require the reviews to be ordered in some meaningful way**. The order of the nodes in the list is not important, and we know that **inserting a node at the head is the least amount of work**.

Let's see how we can implement the **insert_at_head()** function by looking at an example of inserting a couple of nodes into an initially empty list.

Example 3.15 The diagram in Fig. 3.28 shows how the linked list grows as we add new nodes **at the head** of the list.

- Initially, the list is **empty**, which is indicated by the fact that the **head pointer is NULL**.
- The **first node to be added** to the list is a special case. Since the list is empty, the first node added to the list becomes the **head**. This is easily done, we just need to copy the **address of the new node** to the **head pointer** for the linked list.

- Thereafter, any new node can be added to the list by the following two-step process: 1) **Copy the address of the current head of the list** into the **next item pointer** in the **new node** - which links the new node to the rest of the list (at the start of it). 2) **Update the head pointer** to have the address of the **new node** - this makes our list start with the newly added item.

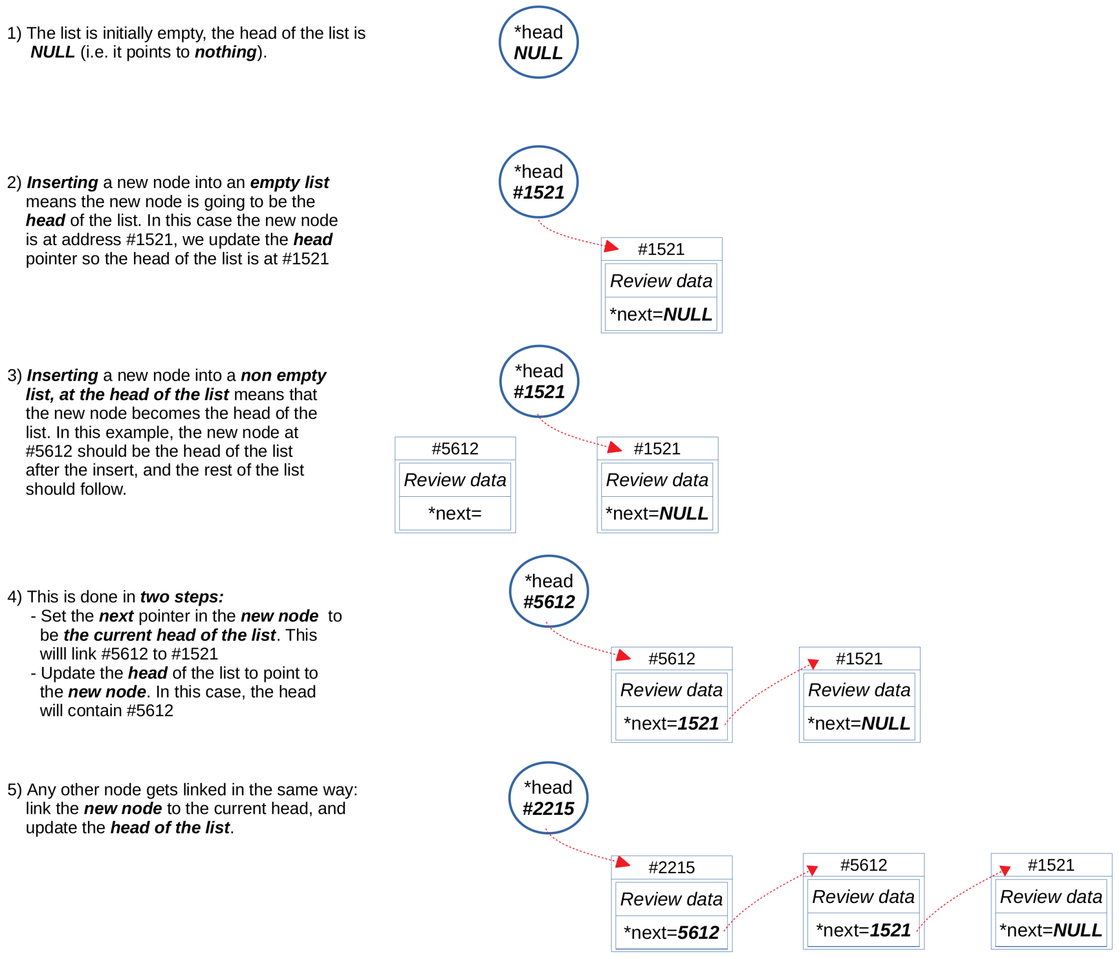

1) The list is initially empty, the head of the list is *NULL* (i.e. it points to *nothing*).
*head NULL
2) ***Inserting*** a new node into an ***empty list*** means the new node is going to be the ***head*** of the list. In this case the new node is at address #1521, we update the ***head*** pointer so the head of the list is at #1521
*head #1521
#1521
Review data
*next=NULL
3) ***Inserting*** a new node into a ***non empty list, at the head of the list*** means that the new node becomes the head of the list. In this example, the new node at #5612 should be the head of the list after the insert, and the rest of the list should follow.
#5612
Review data
*next=
4) This is done in ***two steps:***
- Set the ***next*** pointer in the ***new node*** to be ***the current head of the list***. This willl link #5612 to #1521
- Update the ***head*** of the list to point to the ***new node***. In this case, the head will contain #5612
*head #5612
*next=1521
5) Any other node gets linked in the same way: link the ***new node*** to the current head, and update the ***head of the list***.
*head #2215
#2215
*next=5612

Figure 3.28: Sequence showing how the linked list grows as nodes are added **at the head** of the list.

.

Note

Ensuring that the last node in the list has a *next pointer* that is *NULL* is crucial. If it contains junk, or a previous pointer value, then any program using the linked list will believe there are more nodes and go looking for them at whatever address it finds in the *next pointer*. This is a bad type of bug – it will produce unpredictable behaviour or, if you're lucky, crash your program. If you are seeing weird behaviour in code that uses linked lists, check that the lists are properly ended with a *NULL* pointer at the tail.

✍ **Exercise 3.10** Starting with the diagram in Fig. 3.28 show what the list would look like after we insert two more reviews, the first node added has the address **#3141**, and the second one has the address **#9811**.

Having understood how the insertion process works, let's write a function to insert a new node into the list:

```
Review_Node *insert_at_head(Review_Node *head, Review_Node *new_node)
{
     // This function adds a new node at the head of the list.
     // Input parameters:
     //       head: The pointer to the current head of the list
     //       new_node: The pointer to the new node
     // Returns:
     //     The new head pointer

     new_node->next=head;
     return new_node;           // Same thing as doing
                                // head=new_node;
                                // return head;
}
```

As you can see, it's a pretty short function! - It copies the current head node's address to the new node's **next pointer** (using the **'->'** operator because **new_node** is a pointer). Then it **returns the address of the new head node** – which is the same as the address stored in **new_node**.

Note that we do not explicitly check for an empty list! so as an **exercise** - trace the code in function **insert_at_head()** and convince yourself that it works also when the list is initially empty!

✍ **Exercise 3.11** Draw a memory model that shows what happens when we call the **insert_at_head()** function from **main()** with the line

```
    head=insert_at_head(head,new_node);
```

Assume the list is initially empty and that we requested a new node and got its address in **'new_node'**. Your diagram should show

- The **head pointer** variable in main().
- The **new_node** pointer variable in main().
- The **box for the new node**, somewhere in memory.
- All the variables, input arguments, and return value for **insert_at_head()** just before the memory for this function is released.
- The **values for all pointers** once the call to **insert_at_head()** returns.

Once you have completed the above, draw **another memory model** that shows what happens **when the next node is inserted** into the list.

✍ **Exercise 3.12** Implement a function **insert_at_tail()** that adds a new review to the linked list, but **at the tail** of the list. You will need to add an option to **main()** to allow the user to select this function.

✍ **Exercise 3.13** Given the memory models you created for Exercise 3.11, think about how **you could verify** that both of the insert functions (at head, at tail) are doing the right thing. You want to think about this as a problem in **checking that the list is correctly organized in memory** given the sequence of nodes that have been added to it, and the type of insert function that was used. You should write down **a sequence of insert operations your program will carry out on the list**, and for each of these operations, **how to verify** that the result is correct.

Embedded in the above is the idea that we **test and verify our program as we develop functionality**. A topic that we will explore in more detail shortly!

That completes option **1 - Add a new review**. Let's see now how we could implement option **2 - Print existing reviews**.

The process we have to carry out for implementing option **2** is one of the most common operations you will have to do with linked lists: **Traversing the linked list** while carrying out some particular operation at each node. The operation here is simply printing out the contents, but in more complex applications, where your linked list contains all kinds of complex information, the operation itself may be fairly involved. Regardless of what operation is being carried out, the list traversal process is identical. Let's have a close look at how it works.

3.7.2 Traversing a linked list

The process of **traversing** a linked list requires you to:

- Set up a **traversal pointer** that will be **updated as we travel** down the list to point to the node currently being processed
- **Initializing** the traversal pointer to the address of the **head node** for the list
- Writing a loop that: 1) **Processes** the node whose address is in the current **traversal pointer**, 2) **Updates the traversal pointer** to point to the next node in the list. The loop ends when the **traversal pointer** reaches the **end of the list**

Let's apply the above to write a small function that prints out all the reviews in the list.

```c
void print_reviews(Review_Node *head)
{
  Review_Node  *p=NULL; // Traversal pointer

  p=head;        // Initialize the traversal pointer to
                 // point to the head node

    while (p!=NULL) // Until we get to the end of the list
  {
      // Print out the review at this node
     printf("Restaurant Name: %s\n",p->rev.restaurant_name);
     printf("Restaurant Address: %s\n",p->rev.restaurant_address);
     printf("Restaurant Score: %d\n",p->rev.score);

     // Update the traversal pointer to point to the next node
     p=p->next;
  }
}
```

This deserves a bit of thought. Let's see how it works with the example linked list from example 3.15. The process is illustrated in Fig. 3.29.

Question: What happens if we pass an empty list to the print function? Does it work just fine or will it crash our program?

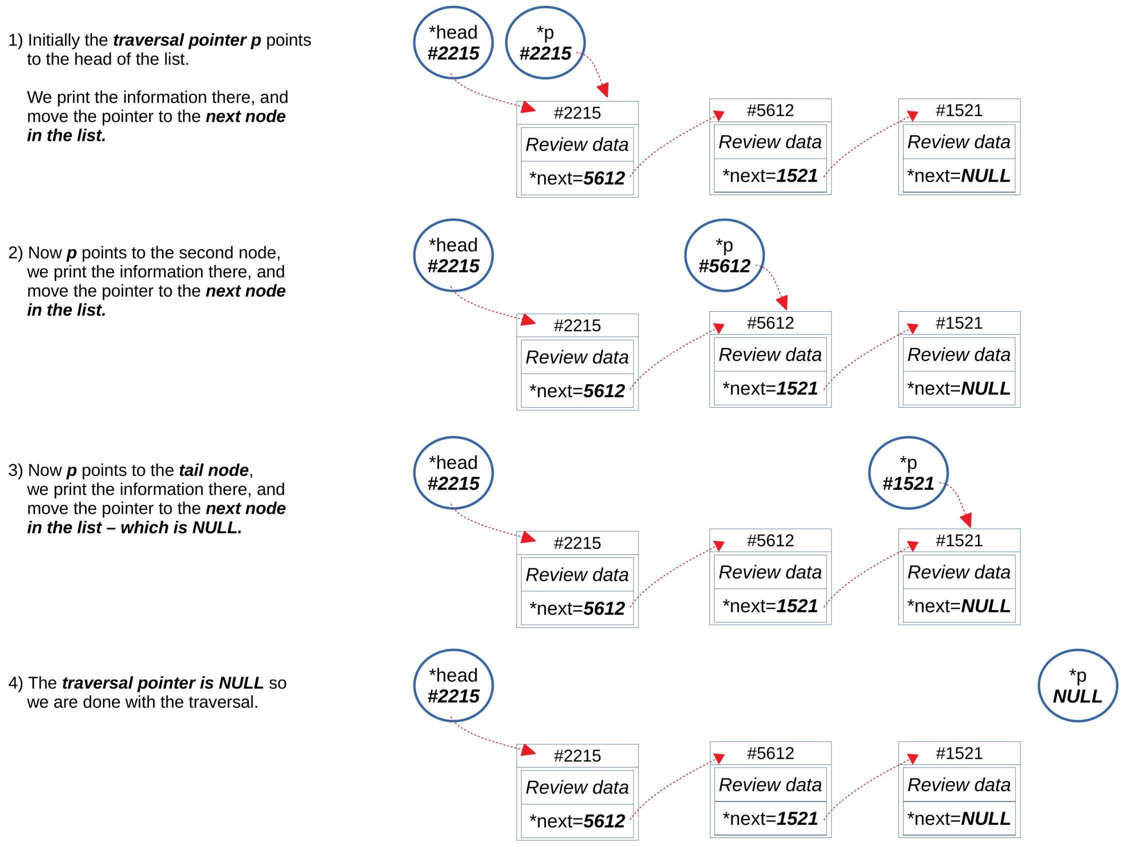

1) Initially the ***traversal pointer p*** points to the head of the list.
We print the information there, and move the pointer to the ***next node in the list.***
2) Now ***p*** points to the second node, we print the information there, and move the pointer to the ***next node in the list.***
3) Now ***p*** points to the ***tail node***, we print the information there, and move the pointer to the ***next node in the list – which is NULL.***
4) The ***traversal pointer is NULL*** so we are done with the traversal.
*head #2215
*p #2215
*p #5612
*p #1521
*p NULL
#2215
Review data
*next=5612
#5612
Review data
*next=1521
#1521
Review data
*next=NULL

Figure 3.29: Sequence showing how the **traversal pointer p** moves down the linked list, visiting each node in sequence until reaching the end of the list (indicated by a **NULL** pointer).

.

The final part of our little program involves option **3**, when the user wishes to exit. This would be trivial were it not for the little detail of **releasing all the memory we have requested** for reviews in our list.

As it turns out, **releasing memory** for a linked list **is just another list traversal** process like the one we discussed just above, except in this case, **instead of printing** the contents of the node, **we will release the memory** allocated to that node. Here's a little function that cleans up after our program:

```
void delete_list(Review_Node *head)
{
  Review_Node *p=NULL;
  Review_Node *q=NULL;

  p=head;
  while (p!=NULL)
  {
       q=p->next;
       free(p);
       p=q;
  }
}
```

The **delete_list()** function above is a bit different from **print_reviews()** function. Notice we are using a **temporary pointer 'q'** which did not appear in the function that prints information. This is required because when releasing memory, we get into a **chicken and egg** kind of problem:

- We need to release memory allocated to the node currently indicated by the **traversal pointer p**.
- But we then need to access the **next pointer** in order to know where the next item in the list is.
- But we can not do that if we already released memory for the node.
- So the **temporary pointer 'q'** is used to store the value of the **next pointer** before we release the box where it is stored.

3.7.3 What have we accomplished up to this point?

By now, we know how to build a linked list to store items for any data type. This is a big deal - there is a huge number of applications out there that rely on linked lists to organize and process information. We will find linked lists in a variety of flavours, and in different programming languages. But they all follow the process described above, and are organized in the same way as the lists we studied in this Section.

You should be comfortable with the code for the program we developed above. Make sure you understand what is going on at each step, and how each of the functions there works. A good way to check your understanding is to explain how the code works to someone else, or to write a summary in your own words, for yourself, explaining what the code is doing and why.

What's next? There are two major operations on linked lists that we have yet to learn: **searching** for a specific item, and **deleting items** from the list. Let's have a look at those to complete our study of linked lists.

3.8 Searching for specific items in large data collections

We started this section with the goal of understanding how to **organize, store, and manipulate a large collection** of information. Perhaps the most important aspect of doing this is **being able to search** through a collection of data **for items of interest**. Consider how many times in the past month you have:

- Used Google to look for a document, class notes, news, or images
- Used the search function in an on-line retail shop to find an item you wanted to buy
- Searched for a particular music video by song title, or by artist name

A very large number of real-world applications have a built-in search function that allows us to find and explore specific data items stored within a large collection. To a large degree, the usefulness of these applications is tied to how efficiently and accurately they are able to find the information a user needs.

In computer science, a very large effort has been invested in figuring out what are the **optimal ways to organize information** so that we can **quickly search through very large collections**. In this Chapter, we will begin looking at this problem, see how far we can get with linked lists, and understand just how much work is needed to search through a large collection that has been organized as a linked list.

This will open the door for us to start thinking in terms of the **efficiency of a particular algorithm, or a particular data structure**, and thus allow us to choose between different data structures that implement the same **ADT**, and/or between different **ADTs**, and select the one that provides the best performance (and as we will see, the definition of performance depends on what we want to achieve with our program).

3.8.1 Searching through a linked list

The **search** process on a linked list **is just a form of list traversal**. We have already seen how a list traversal works, and **the only difference** when we are searching is that the operation carried out at a node **is a comparison** between a **search key**, and a **value or set of values stored in the list node**. The search process will either

- Find the requested search key and **return a pointer to the node that contains it**, or
- Go through the entire list without finding the key, and return **NULL**

Let's see how we would write a search function for our linked list of restaurant reviews so that we can update the score of a specific restaurant already in the list. The search function should accept a **restaurant name as search key**, and **return a pointer to the node** that contains the review for that restaurant, or else return **NULL** to indicate there is no restaurant with that name in our list.

```
Review_Node *search_by_name(Review_Node *head, char name_key[])
{
  // Look through the linked list to find a node that contains a
  // review for a restaurant whose name matches the 'name_key'
  // If found, return a pointer to the node with the review. Else
  // return NULL.

  Review_Node  *p=NULL; // Traversal pointer

  p=head;
  while (p!=NULL)
  {
    if (strcmp(p->rev.restaurant_name,name_key)==0)
    {
      // Found the key! Return a pointer to this node
        return p;
    }
     p=p->next;
  }
  return NULL; // The search key was not found!
}
```

The code above looks through the linked list. At each node, it compares the restaurant name in the review stored at the node with the search key, and if they are equal, it returns the pointer to that node.

Note

This is one example of code in which it makes perfect sense to exit a loop early – as soon as we find the search key we return the pointer to the node where we found it. Imagine a list with millions of entries, it would make no sense to keep traversing each of those nodes after we have found what we were looking for.

We can now modify our original program - the one that handles restaurant reviews, so that it allows the user the option of updating a review that has already been added to the list. This requires us to change a bit the option listing in **main()**:

```
    printf("Please choose one of the following:\n");
    printf("1 - Add a new review\n");
    printf("2 - Print existing reviews\n");
    printf("3 - Update review for one restaurant\n");
    printf("4 - Exit this program\n");

    scanf("%d",&choice);
    getchar();
```

We need to update the **while loop** that prints these options and requests a number from the user, so that it exits when the user selects **4** (previously it was **3**), and we need to add code to use our search function to look for a specific restaurant, and update its score:

```
  else if (choice==3)
  {
    printf("Which restaurant's score do you want to update?\n");
    fgets(name,MAX_STRING_LENGTH,stdin);
    one_review=search_by_name(head,name);
    if (one_review==NULL)
    {
      printf("Sorry, that restaurant doesn't seem to be in the list\n");
    }
    else
    {
      printf("Please enter the new score for the restaurant\n");
      scanf("%d",&one_review->rev.score);   // Store the new score directly in the review node!
      getchar();
    }
  }
```

Adding these improvements to our code allows us to update reviews for restaurants already added to our linked list.

✍ **Exercise 3.14** Compile and run the complete program (the full listing can be found in Section 3.14, you can copy/paste it into a file), insert a few reviews, print the reviews in the list, and then modify one of the reviews. **Be sure to test what happens when:**

- You try to print a list that is **empty**
- You try to **update** a review for a restaurant that **does not exist**
- You choose an option not in 1-4 when prompted

Try to break the program. See what you can do to **make it act weirdly or crash**, and then **think about how you would prevent a user from breaking the program** in that way.

With this we are taking our first steps into **software testing**, an incredibly important process that has to be carefully and thoroughly carried out for every piece of software that we develop. We will explore it in more detail at the end of the Chapter.

✍ **Exercise 3.15** Write a search function **search_by_address()** that allows you to modify a restaurant's score by searching for that restaurant's address. Add an option to the menu in **main()** to use your new function, and implement the code that updates the score. **Test your code** and make sure it's solid, **updates the correct review**, and **doesn't break if the user enters a non-existent address**.

Question: Should we do something to ensure the **score** entered by the user is valid?

✍ **Exercise 3.16** Write a **search function** that prints out information for **all restaurants with a review score greater than, or equal to a user-specified value**. Add an option to the menu in **main()** to allow the user to select this functionality. **Test your code** and make sure it's solid, prints out all the restaurants in the list that meet the specified criteria, and **doesn't print** any restaurants that don't meet the criteria.

Question: Suppose we are searching for a specific restaurant by name in a list with 1,000,000 nodes. In the worst possible case (i.e. if we are unlucky and have to do the longest possible traversal), how many nodes will we have to examine before we find the desired restaurant or determine it's not in the list? How does that number change if the list has 10,000,000 nodes? What about 100,000,000 nodes?

What we should learn from the above:

- **Search** on a linked list is just a **list traversal** checking each node to see if it contains the **search key**.
- Because it is a list traversal, we may have to **go through the entire list** looking for a node.
- This means the **amount of work** we have to do during search **grows with the length of the list**.
- The amount of work **search** has to do in the **worst case** grows in **direct proportion** to the **number of data items in the list**.

The next Chapter will explore in detail how we can **estimate**, **quantify**, **compare**, and **think about** the amount of work a particular algorithm has to do when working on a collection of a given size. This will be a central concern for us as we discover more and more **ADTs** and **algorithms** many of which can be applied to the same problem.

We will need a **principled way** to choose **the most efficient** one (under a definition of **efficiency** that is general and does not depend on particular implementations, languages, or hardware).

However, just from the discussion above it should be clear to us that for a very large collections of data (e.g. the millions upon millions of documents indexed by Google), a **linked list will simply not be a fast enough data structure** to allow users to frequently and quickly find the information they need. Imagine how long it would take if every time you input a search keyword in Google it had to go through a list billions of nodes long looking for it!

We will need a faster way to do search through large collections. Unfortunately, there's little we can do to make searching on linked lists faster, so we'll have to come up with smarter data structures. That's a little later on though. For now, let's set down a few more important thoughts related to search that will have importance later on (for example, if you choose to spend time studying and working with databases).

3.8.2 Thoughts on search

We should spend a bit of time thinking about the **search key** we are using to look through our list of reviews.

Question: What should be the **properties of a good search key?**

Think about the restaurant name. At first glance this may look like a good choice and it worked for our little program with a few reviews entered by a single user. However, consider the following:

```
- What if the user typed in "MacDonald's" as a search key?
  * Would we expect there to be a single node for MacDonald's?
  * Would we expect to find multiple entries? (e.g. one for each different location, each
     with a different address)
  * If there are multiple matches for a specific search key what should the program do?
    Update them all one by one?
    Ask for more information to single out one location?
    Give up and refuse to update?
```

The point to make here is that though we can search for information using any field, **a good search engine will have a way to uniquely identify each entry in a collection.**

One of the fundamental tasks that have to be carried out when designing a database, is figuring out what **data items** it will store, for each of these items, **what data fields** are required to store the relevant information, and **the list of search keys** that can be used to find information within the database.

Of particular importance, we will require **at least one primary key**. A primary key is either a single data field, or a collection of fields such that they **uniquely identify** each individual item in the database. For example, **a social insurance number** can be used to uniquely identify an individual, regardless of how many different people may have identical names. Most serious applications provide a **unique numeric or alphanumeric identifier** - such as the social insurance number, a student number, an employee number, passport number, etc. to be used as **primary key** while looking for information for a particular individual.

In the case of our restaurant review database, we do not have any data field that can serve as a unique identifier. However, we can **create a unique identifier by combining multiple fields**. For example, we can use **the restaurant's name** together with **the restaurant's address**, and ask the user to provide **both** of these when searching for a particular review. The reason why this works is that we would not expect two different restaurants to have exactly the same name and be located at exactly the same address (if we find such a case, it means we have a duplicate entry in our collection!).

Collections are made up of unique items, **no duplicates** are allowed. This is usually enforced by **having the insert function check** whether or not a particular data item has already been added to the collection, and **refuse to insert duplicate items**. In the case of our restaurant review application, the insert function would check that the collection doesn't already contain a review with the same restaurant name and address.

3.9 Deleting nodes from a linked list

The last operation we need to implement to complete our linked list is the **delete** or **remove** operation. As the name implies, it removes a specific node from the list. Because **it looks for a specific item**, it involves a slightly modified search process – so **it is in essence a list traversal operation**.

Let us have a look at the steps needed to delete a node.

Example 3.16

Figure 3.30 shows the different cases for **deleting** a node, and what the steps are depending on where the node to be deleted is on the list. The process is:

- Case a) - Deleting the node **at the head** of the list. Two-step process: update **head=head->next;** (which moves the head pointer to the second node in the list), then **delete** (which means use **free()** on) the old **list head**.
- Case b) - Deleting the node **at the tail** of the list. Three-step process: **traverse the linked list** until we reach **the node that is second from last** (this will become the new **tail** of the list), **delete** the **current tail**, then set the **next pointer** in the **new tail node** to **NULL**.
- Case c) - General case, the node we are deleting **is in between two other nodes**. Three-step process: **traverse the list** until we reach the node **immediately before** the one being deleted (the predecessor of the node being deleted). Update the predecessor's **next pointer** to contain the address of the node **immediately after** the one being deleted (the successor of the node being deleted). Finally, **delete** the node we want to remove.

These cases are illustrated in the figure, you should write an example of a list for yourself, and try out all three cases to verify you can easily figure out which pointers need to be updated for each case.

There is more than one way to implement the deletion operation. The key here is that for cases **b)** and **c)** in Example 3.16 we need **to keep track of the predecessor** of the node that is being removed. We can achieve this in one of two ways:

- 1) Use **two** traversal pointers. One for the current node, and one that is always right behind it along the list and points to the **predecessor**.
- 2) Use **one** traversal pointer, together with the current node's **next item pointer** to **look ahead** for the node we are deleting (or the **tail** of the list).

Both ways work, and you should use whichever makes more sense to you. A sample implementation of the delete function is shown below:

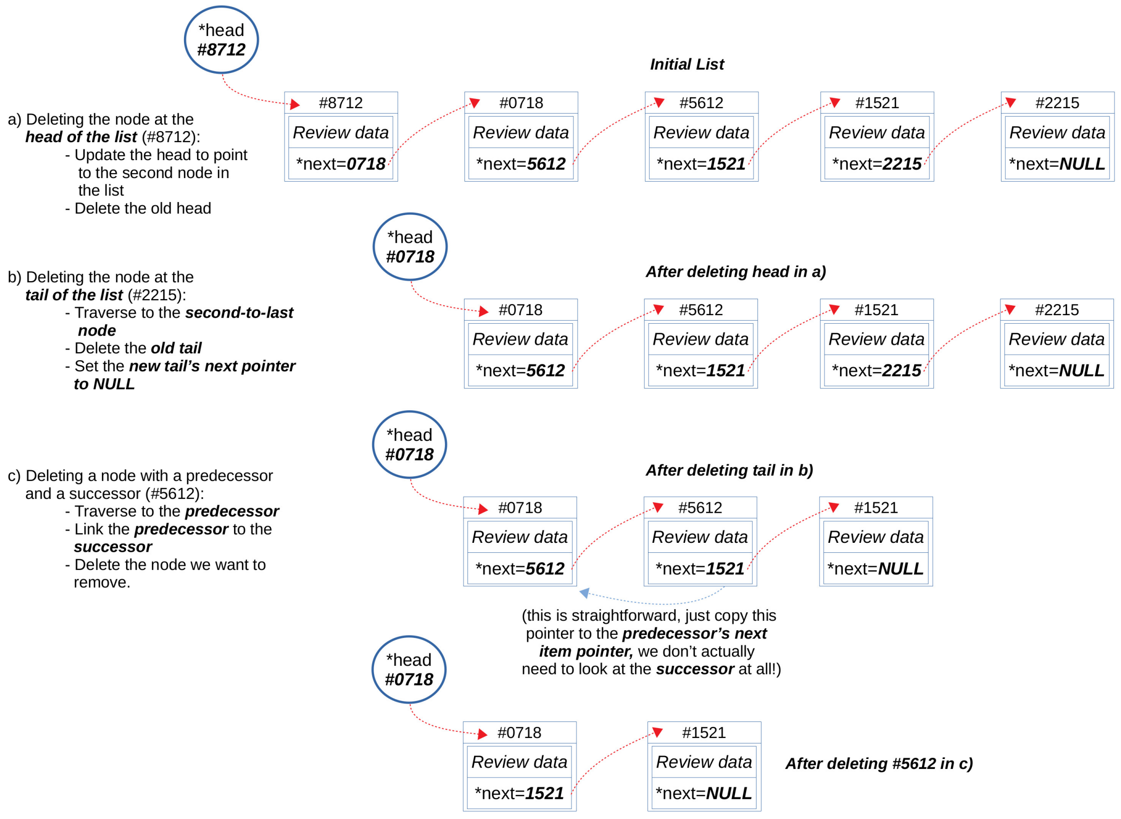

*head
#8712
Initial List
a) Deleting the node at the head of the list (#8712):
- Update the head to point to the second node in the list
- Delete the old head
#8712
Review data
*next=0718
#0718
Review data
*next=5612
#5612
Review data
*next=1521
#1521
Review data
*next=2215
#2215
Review data
*next=NULL
*head
#0718
After deleting head in a)
b) Deleting the node at the tail of the list (#2215):
- Traverse to the second-to-last node
- Delete the old tail
- Set the new tail's next pointer to NULL
*head
#0718
After deleting tail in b)
c) Deleting a node with a predecessor and a successor (#5612):
- Traverse to the predecessor
- Link the predecessor to the successor
- Delete the node we want to remove.
(this is straightforward, just copy this pointer to the predecessor's next item pointer, we don't actually need to look at the successor at all!)
*head
#0718
After deleting #5612 in c)

Figure 3.30: Three different cases for deleting nodes from the list, and what the list looks like after each node has been correctly deleted.

.

```
Review_Node *delete_by_name(Review_Node *head, const char name_key[])
{
  // This function removes the node from the link list that contains the
  // review with a matching restaurant name.

  Review_Node *tr=NULL;        // We will use the two pointer process
  Review_Node *pre=NULL;

  if (head==NULL) return NULL; // Empty linked list! nothing to do!

  // Set up the predecessor and traversal pointers to point to the first
  // two nodes in the list.
  pre=head;
  tr=head->next;

  // Check if we have to remove the head node - case a)
  if (strcmp(head->rev.restaurant_name, name_key)==0)
  {
       free(pre);  // Delete the first node in the list
       return tr;  // Return pointer to the second node (new head!)
  }

  // This while loop takes care of cases b) and c)
  while(tr!=NULL)
  {
      if (strcmp(tr->rev.restaurant_name, name_key)==0)
      {
        // Found the node we want to delete
        pre->next=tr->next;     // Update predecessor pointer
        free(tr);               // remove node
        break;                  // Done!
      }
      tr=tr->next;
      pre=pre->next;
  }
  return head;   // Head did not change
}
```

The code above implements the list traversal using **two pointers**, a **predecessor pointer** and a **traversal pointer**. It first checks whether we're deleting the head node and if so, it returns the updated head node pointer. Otherwise it proceeds through the loop that finds and removes the node that contains the specified search key (if such a node can be found in the list).

✍ **Exercise 3.17** Compile and run the program (the full listing can be found in Section 3.14), and **test** the **delete_by_name()** function to verify it works correctly.

As we have discussed earlier, you need to

- Come up with a sequence of linked-list operations that will thoroughly test the delete functionality.
- Perform the operations in your sequence one by one, checking after each of them that the structure and contents of the list are correct.

Question: How can we check that a linked list in the computer's memory has the correct structure?

This is indeed the central problem we have to solve any time that we are **testing code** for correctness. It always boils down to this: **we must know what the expected situation would be after our program performs some**

operation. For example, we can draw **on paper** a diagram of the linked list, and figure out how the nodes are linked as we insert and delete nodes from the list.

We then have to check that the list **that our program built in memory** has the expected structure (i.e. nodes are in the correct order, and linked as expected). To do this we need our program to provide us with information about **what is stored at each node, and how the nodes are linked**.

The easiest way to do this is to write a small function that prints out the information we need, then call this function after each operation and check the information printed against our hand-written diagram.

In the next Chapter, we will discuss how to use more sophisticated tools called **debuggers** that will allow us to **stop the program at particular points** (where we want to check something), **inspect what is stored in variables and dynamic memory** (to check that the information is correct), and if needed **change values of memory contents** which may be useful to test particular parts of our code, or see what happens if variables take on certain values that may cause trouble.

For now, printing the information we need will allow us to verify what the program does against our work on paper, showing what the list should contain after each operation.

✍ **Exercise 3.18** Re-implement the **delete_by_name()** function so that it uses **one traversal pointer**, and instead of a **predecessor** pointer, it uses the **next item pointer** in the current node to **look ahead** for the node we want to delete.

Test the re-implemented function, ideally **using the same tests** you developed for the previous exercise. Verify that the function correctly deletes nodes, and that the list has the correct ordering and is linked as expected after each deletion.

✍ **Exercise 3.19** Write a delete function that allows a user to **delete reviews by specifying the restaurant address**. **Test** the function for correctness.

✍ **Exercise 3.20** Write a delete function that allows a user to **delete all reviews that have the specified score** (e.g. we may want to remove from our list all restaurants with really bad scores, such as 1). **Test** the function for correctness.

All that remains to complete our little program for storing, organizing, and updating restaurant reviews is to add one more option to main() allowing the user to delete a review for the specified restaurant. This also completes our study of linked lists in **C**. It is a very common data structure, and any implementations you find in the future that are written in **C** will be very similar to what we covered above.

3.10 Queues

An important variation on the **List ADT** is the **Queue ADT**. A queue is simply a list where the **insert/add** and **delete/remove** operations happen at very specific locations in the list.

- In a **queue**, items are **always added at the tail** of the queue - this operation is called **enqueue**
- In a **queue**, items are **always removed from the head** of the queue - this operation is called **dequeue**

In addition, other operations are often defined as well, for example, getting the **length of the queue**.

Queues are ubiquitous (they appear everywhere!). For instance, a network printer will have a **print job queue**. Print jobs can arrive at any time, from any of a possibly large number of users. Jobs are printed in the order in which they arrive.

Queues are also important in software that simulates **scheduling operations**. For example, Air Traffic Control software will have queues for departing flights, inbound flights, and aircraft that are slotted for landing.

Online customer service systems often have a **wait queue** to which arriving users are added.

Finally, queues are essential for applications that use **graphs** to represent and process information. Graphs are **ADTs** that represent collections of information in terms of **data items** and **their relationships to one another**. One common example are **social networks**, these have nodes for users, and the links indicate connections between users (see Fig. 3.31). We will study **graphs** in detail in a later Chapter.

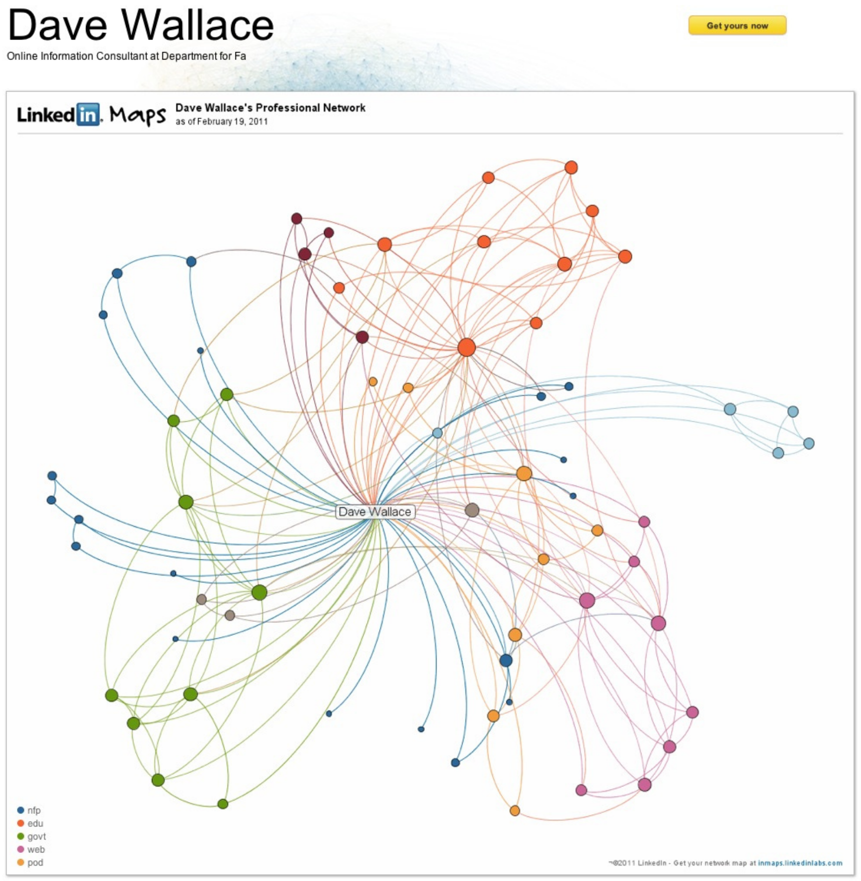

Dave Wallace
Online Information Consultant at Department for Fa
Get yours now
LinkedIn Maps
Dave Wallace's Professional Network
as of February 19, 2011
Dave Wallace
nfp
edu
govt
web
pod
©2011 LinkedIn - Get your network map at inmaps.linkedinlabs.com

Figure 3.31: This image shows a graph for the professional connections of a single individual. Each **circle** represents an individual, and each **line** represents a professional connection between two individuals. *Image: Dave Wallace, Flickr, CC-SA 2.0*

Processing information in graphs often involves placing nodes in a queue. Common instances of this include the classical **Artificial Intelligence** search methods that do path-finding (think about the Maps application in your cellphone), scheduling, finding solutions to constrained optimization problems, and so on.

You will have a chance to explore many more applications of queues, so do not forget what the **Queue ADT** looks like. Importantly, **you already know how to implement a Queue ADT!** It can be implemented using a

linked-list in which the **insert operation** always adds nodes at the **tail**, and the **delete** operation **always removes the head** of the list.

✍ **Exercise 3.21** Extend the liked list implementation we developed in this Chapter so that it supports using the list as **Queue ADT**. You should implement an **enqueue()** function, and a **dequeue()** function, and provide options for the user to select these operations from the menu in **main()**.

As ever with exercises for the rest of the book: **thoroughly test** your **queue** and make sure that it always has the correct structure, items are in the right place, and links are correct at all times.

3.11 Wrapping up and summary

After working our way through this Chapter, carefully thinking through the examples and programs developed here, and completing all the exercises, we should:

- Be able to explain why we need to think carefully about how to store and organize collections of data.
- Be able to explain why we need to be able to reserve space for data items on-demand - i.e. we should know what the limitations of arrays are and why they are not a good solution for applications where the amount of data is not specified at the start and changes over time.
- We should know how to create compound data types (bento boxes!) for data so we can represent complex items.
- We should know how to create and manipulate variables and pointers for complex data types.
- We should know what a **List ADT** specifies, how it organizes data, and what operations it supports.
- We should understand how a linked list works, conceptually. How to search for an item, insert an item, and delete an item in a linked list.
- We should be able to implement a linked list data structure in C. including:
  - Defining compound data types to hold the information we need
  - Defining a node data type that we can use to build the list
  - Implementing the insert function, at head, at tail, or in-between nodes
  - Implementing the search function to find and update specific items
  - Implementing the delete function to remove nodes as needed
  - Releasing memory we allocated to nodes in the list
- We should understand how much work is involved in list traversal. We should be able to explain why we say that the amount of work for traversal is proportional to the number of items in the list.
- We should be able to explain the difference between an **ADT** and a **data structure**.
- We should be able to go through the program listing in Section 3.14 and understand everything that it is doing.
- In addition to that we should have learned the specifics of the **C** implementation that we used to build the restaurant reviews app:
  - How to read input from the terminal
  - How to allocate memory for a list node on-demand using **calloc()**
  - How to write a little driver program that allows us to test parts of our code

- How and when to pass and return pointers so functions can work on linked lists

3.11.1 Why this Chapter is important

We started this section needing a way to:

- Store, organize, and manage a possibly large collection of complex data items
- Be able to obtain space for data items on-demand
- Be able to search for specific items, and to grow or shrink our collection as needed

We discussed the idea of **containers**; then we looked at a particular container type - the **List ADT**. We saw how to use a **linked list** to implement a **List ADT**, and we spent time working out the implementation of a **linked list data structure**.

We now have a working linked list implementation, and we can create variations of this list to handle pretty much any data type we may ever need to store and keep organized. This is our first achievement for this part of the book.

Indeed, linked lists or variations of them will appear on a large majority of applications. To give you a couple examples you may find amusing:

- Graphics rendering programs keep lists for most data items used to create images: objects to be rendered, light sources, textures, animation key-frames, etc.
- Music synthesizers keep a list of notes being played, to be fed to the sound synthesis engine. There are also lists of digital effects, and even entire songs kept in a list in memory for playback.
- A shopping cart for an on-line retailer can quickly and easily be implemented with a linked list.

These are only a couple applications, there are many, many more. However, at this point we also know that **linked lists** have the disadvantage that search (and thus updating information for specific nodes), deletion, and list traversal take a fair amount of work - we may have to go through the entire list checking each node in turn to find what we want.

This means that for applications that will handle very large amounts of data, linked lists would result in an unacceptably long wait for basic operations that need to be carried out thousands of times.

So, while lists provide us with a way for satisfying the data organization and storage goals we set out to fulfill at the start of the section, we now know we need to find a smarter way to organize data if we want to ensure the fastest possible access to possibly very large amounts of information. We also need to figure out a principled way to study, and compare the amount of work that is done by different algorithms or data structures as they process information.

These will be the topics of the next Chapter in the book. For now, let's see what kinds of problems we can solve having learned about containers and lists!

3.12 Problem Solving

As we said at the start of the book, our goal is to learn general techniques for solving problems in computer science. In this Chapter, we learned about containers and lists. Our motivation in doing this was the need to understand how we can organize, store, and manage a possibly large amount of complex information so as to make it useful within a program.

The problems below give you a chance to test your understanding of the material in this section, and to practice problem solving. But first, let's look at a reasonable approach you may want to consider when faced with a new problem.

3.12.1 A suggested approach to solving programming-related problems in CS

- **Read** the problem description **carefully**. If there is something in the problem's statement that is unclear, seek additional information - this may require a bit of research.
- **Consider the input** for the problem - that means, what data will you be working with, whether you know how much of it there will be from the start, or whether the amount will change (and likely grow) over time, as well as any particular characteristics of the input data. Consider as well whether user input will be required.
- Write down any assumptions you are making about the data:
  - Data types you think will be needed, new compound types you'll have to create
  - Special conditions (e.g. range of input values, or description of valid inputs)
  - Uniqueness constraints (e.g. If an input field contains values that must be unique, such as student numbers)
  - Amount of data you may expect to deal with
  - What kind of storage structure you think will be needed (e.g. arrays vs. lists)
- Consider the task the problem requires you to solve: In order to find a good programming solution, you need to understand what will happen to the input data once it's in your program, make a note of what operations or processing will be performed on the input data, and whether it will be applied to all or most of the data or individual items.
- **Consider the output** for the problem: This means thinking about what needs to be computed or produced by your solution. Is the output used only for display (e.g. to be printed to the terminal), or is it going to be the input for a different part of a program. Depending on this, you need to think about how to store the output. Write down your assumptions about the output.
- Write down the solution in plain language (not code). At this point you want to make sure you understand the solution for the problem and can think of every step involved
- Design and implement the solution. The design must be informed by your analysis of the input and output to the program, as well as what processing will be done on the input data.
- **Test** your solution thoroughly, make sure it solves the problem with reasonable input. That may involve running your code multiple times with different possible inputs, carefully chosen to cover different possible but valid inputs to the program. It must work every time.
- Address any issues discovered during testing.

- **Test** your solution for **special cases**, e.g. empty or missing input values, input that is the wrong data type, input that breaks your initial assumptions about the data (that's why we wrote them down!). Resolve any issues identified in testing.
- Now **try to break the program**. See if you can come up with input that causes your program to crash or do the wrong thing. This may include invalid input, empty fields, using special characters, and so on. The goal here is to identify potential problems, and think about how to make your solution more robust.
- Finally - if the output of your solution is going to be used as input for a different part of the program, **Test** that the output is properly formatted and can be accessed by whichever functions need it.

The process is important - hacking away at a solution without having fully understood the problem will most likely

- Make it harder for you to come up with a good solution.
- Make you think **C** is difficult - because you're having a hard time implementing the solution, the problem is not the language, it's the fact you haven't fully thought through what the solution should be.
- Produce solutions that are of lower quality.
- Produce a solution that is less organized, and is harder to test and maintain.
- Produce a solution that needs to be re-worked because it doesn't do what it's supposed to do.
- Lead to code that is fragile, and easy to break.

Get used to working through a solution methodically, and **thinking carefully** about every aspect of your solution **before you start coding**. Remember: **Being able to come up with a solid, well thought solution to a problem is much more valuable than just being able to implement a solution someone else developed**.

The problems below are intended to make you think, and tohelp you identify what material you still haven't mastered - **do not stress** if they seem challenging. They are meant to be, but with a bit of work and focused studying you will be able to think of a way to approach, and eventually solve every one of them.

3.13 Problems involving containers and lists

Problem 3.1 In practice, we often need to find out the length of a linked list (we need to know, for example, how many restaurant reviews we have in our system at a given time).

Write a small **C** function that takes as input the head of a linked list, and returns the length of the list (zero if the list is empty). You can do this using the linked list for restaurant reviews, or use your own linked list.

Problem 3.2 You are working on a **checkout module** for an on-line store's shopping cart. Because typical users will only add a few things to the cart in any one visit, the store's on-line system keeps the items currently in the cart in a linked list. Each **Item_Node CDT** in the linked list contains:

```
Item      item_info;
Item_Node *next;
```

The **Item CDT** contains

```
  int       item_id;        // A unique identifier for each item
  char      name[1024];     // The item's name
  float     price;          // The item's price
  float     discount_pct;   // Discount percentage in [0, .5] (0% up to 50%)
  int       quantity;       // Item quantity in the cart
```

Part a) Complete the definition of the **Item CDT** and the **Item_Node CDT** in **C**. This is basically to practice your grasp of the syntax needed for defining new data types.

Part b) Write down the implementation of a function that computes the total price for items in the shopping cart:

- First write down the steps of the solution in plain language, and check that your solution makes sense, and computes the correct total considering the **quantity**, and **discount_pct** for each item.
- Then write an implementation in **C** for a function that **computes and returns the total price** for items currently in the shopping cart. You may assume the function will take in a **pointer to the head** of the linked list for the shopping cart.
- Thoroughly **test** your solution for correctness.

Problem 3.3 You have found a summer job at the central **Toronto Public Library**. The TPL has been expanding its digital collection that includes eBooks, movies, audio recordings, and photographs. The library's digital collection is stored in a central server, and you have a linked list of items available.

The **Item_Node CDT** for the list is as shown below:

```
typedef struct Item_List_Node{
  int       item_id;          // Unique identifier
  char      title[1024];      // Title for this item
  int       type;             // Type of resource
                              // 0 - eBook, 1 - video,
                              // 2 - music, 3 - photograph

  //  There are many more fields we don't need for this problem

  struct Item_List_Node *next; // Pointer to next node in the list
} Item_Node;
```

Your problem is as follows: Each local branch of the library houses its **own collection** of video, and music (these are in the form of actual DVDs and CDs). They are now seldom accessed since most users would rather access the same content electronically on their handheld devices. So the library has decided to remove from each branch any videos or music recordings that are already part of the central digital collection.

Library personnel have already cataloged the content at each branch, and stored it in a (you guessed it!) linked list.

Part a) Finding duplicate content: Write down the steps of an algorithm that takes as **input two linked lists** of **Item_Nodes** (one for the central digital collection, one for a local branch collection), and **prints out any duplicate** videos or music entries so the duplicates can be removed from the local branch collection.

Be sure to write down any assumptions you are making regarding the fields in the item node. Write your solution in plain language, with **enough detail** that someone else could implement it in **C**.

Once you're satisfied with your solution, write an implementation in C.

Part b) Think about the data representation. Note that whoever designed the data representation for the library's linked list didn't bother to build a separate data type for each item's information. DVDs and eBooks, for instance, will likely require different fields, the **Item_Node** has to be able to properly capture information for all supported types. To do this, the **Item_Node** will have to contain **every possible field** that may need to be captured for all supported media resources. Write down what you believe would be:

- Advantages of representing items in this way
- Disadvantages of representing items in this way

Problem 3.4 You are working on an open source project for a web browser that provides the user with full control over the amount and type of personal information that is made available to websites. One of the key components of any web browser is the bookmarks section. For simplicity, the bookmarks are organized as a simple linked list. New bookmarks are inserted at the head of the list.

However, the user can choose to organize the bookmarks in many different ways. In particular, they may choose to **sort the bookmarks by url** by pressing a button on the browser's main window.

Part a) Implement a function that builds a sorted linked list. Write down the steps required to

- Take as input an un-sorted linked list of bookmarks.
- Create a new linked list where the bookmarks are sorted by url by inserting each node from the original input list into the sorted list at the right location according to its sorting order (you may want to review insertion sort).

Use plain language, but do **make sure your solution is detailed enough** that someone else could implement it in **C**.

Illustrate with a diagram how your solution works.

Now, assuming that the linked list nodes contain:

```
char     url[1024];
Url_Node  *next;
```

Write a function in **C** that takes as input the **head of an un-sorted linked list** of **Url_Nodes**, builds a **sorted linked list** of **Url_Nodes**, and **returns a pointer to the head** of the sorted list. You can assume you have already written a function

```
Url_Node *copyUrlNode(Url_Node *orig);
```

that takes as input a **pointer to a** *Url_Node* and **creates a new node** with the same URL but with the **next pointer set to NULL** (so you can insert it into the growing, sorted linked list). Yes, we are duplicating information at this point!

Part b) More challenging - Implement a function that **takes an un-sorted** input linked list of URLs, and **sorts it without making a new list**.

As in part **a)**, you should write your solution steps first in plain language, draw a diagram to show how the process works on one node of the input list, and finally write an implementation in **C**.

Because you haven't written a full program to work with the linked lists in this problem, you can't **test** your implementation as you normally would. However, **you can still test** your solution! in this case, you can **trace through the steps of your algorithm** (on paper) making sure that each step is doing the right thing, keeping track of a diagram of the linked list(s) as the algorithm is working, and checking that it produces the correct result (at least on paper).

It is important to always walk through an algorithm you just developed, and to make sure it isn't missing steps, and that it does the right thing.

Problem 3.5 You have been hired for a Co-Op placement at the **University Health Network**. Having seen that you learned **C** during your studies, they decided to give you the task of designing the storage framework for a new system keeping track of the sequence of patients to be seen at an emergency room.

You are asked to:

- Develop a **suitable data representation** to keep track of each patient's information as captured by the triage nurse.
- Develop the **storage framework** (decide what data structures to use to organize and access the information), as well as the functionality required to:
  - Add patients as they arrive
  - Remove patients once they have been seen by a doctor, or if they leave
  - Search for specific patients by name
  - Print out a list of patients in the order they expect to be seen by a doctor

Part a) - Designing the data representation for patients. The nurse at the triage station will capture the following information:

- Patient's name (Last, First, and Middle)
- Patient's street address
- Patient's postal code
- Phone number
- Health card number
- Body temperature in degrees Celsius
- A short description of the problem

Design a **data representation model** that would allow your program to organize and store information for one patient. This model will be the foundation of your triage system.

- Show a list of the data fields and their data type. You should justify (explain) why the data type is appropriate to each field. If you added fields beyond what the nurse captured, explain why these are needed and how they will be used.
- Indicate special constraints you can identify for each field (e.g. range of values, uniqueness constraints, etc.)
- Mark which field(s) will be used for searching for specific items, e.g. to remove specific items, or to implement functionality required by the system.
- Write an implementation in **C** of your data representation model - this will require designing appropriate **CDTs**.

Part b) - Design the core of the triage patient management system. Required functionality:

- The system must allow you to add patients as they arrive at triage:
  - Patients should be seen in the order they arrive
  - Unless their body temperature is > 40.5C, in which case they must be seen first
- A nurse must be able to bump a patient to the front of the list at any point if they believe the patient needs immediate attention.
- Patients must be removed from the system once they've been seen, or if they leave (they may, or may not notify the nurse).
- The current list of patients in the order they will be seen is printed to a screen so the triage nurse can keep track of what's happening at all times.
- Nurses must be able to look-up patient data by health card number, and update any data field as needed.

You need to provide an overall description of the solution that clearly shows:

- What data structure(s) you will use, explaining why you chose that particular data structure and why you think it suits the problem well. Be sure to note any possible limitations of your particular choice.
- How you will break your solution into modules that can be implemented as separate functions.
- A pseudocode description of the main function showing what happens:
  - when a patient arrives
  - when a patient is seen or leaves
  - when a nurse bumps a patient to the front of the list
  - when a nurse updates a patient's record because their body temperature has changed
- A pseudocode description of the part of your solution that adds a new patient to the list.
- A pseudocode description of the part of your solution that moves a patient to the front of the list.

At this point you have solved the problem - **that's the important part!** All that remains is implementing the solution and testing it for correctness.

Part c) - Implement and **thoroughly test** your solution!

What you will achieve by solving this problem:

- You'll have gone through the full process of designing and implementing a solution to a data organization/storage/management problem that applies to a real world situation

- You will find any gaps in your understanding of this Chapter's material
- You will practice every concept covered in this Chapter as you are developing your solution
- You will practice implementing **C** code that deals with compound data types, and data collections

3.14 Program listing for the linked list implementation in C

```
/*
   Introduction to CS - Chapter 3 - Containers, ADTs, and Linked Lists

   This program implements a linked list of restaurant reviews.
   The program allows the user to enter as many reviews as needed,
   to print the existing reviews, and when finished, it releases
   all memory allocated to the list before exiting.

   (c) 2024 - F. Estrada, M. Ponce, I. Huang
*/

#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define MAX_STRING_LENGTH 1024

typedef struct Restaurant_Score
{
       char restaurant_name[MAX_STRING_LENGTH];
       char restaurant_address[MAX_STRING_LENGTH];
       int score;

} Review;

typedef struct Review_List_Node
{
  Review rev;
  struct Review_List_Node *next;
} Review_Node;

Review_Node *new_Review_Node(void)
{
  Review_Node *new_review=NULL;  // Pointer to the new node

  new_review=(Review_Node *)calloc(1, sizeof(Review_Node));
    if (new_review==NULL)
    {
        printf("new_Review_Node(): Error! - no memory left, can not create new node!\n");
        return NULL;
    }

  // Initialize the new node's content with values that show
  // it has not been filled. In our case, we set the score to -1,
  // and both the address and restaurant name to empty strings ""
  // Very importantly! Set the 'next' pointer to NULL

  new_review->rev.score=-1;
  strcpy(new_review->rev.restaurant_name,"");
  strcpy(new_review->rev.restaurant_address,"");
  new_review->next=NULL;
  return new_review;
}

Review_Node *insert_at_head(Review_Node *head, Review_Node *new_node)
{
     // This function adds a new node at the head of the list.
     // Input parameters:
     //       head : The pointer to the current head of the list
     //       new_node: The pointer to the new node
     // Returns:
     //      The new head pointer

     new_node->next=head;
     return new_node;
}

void print_reviews(Review_Node *head)
{
  Review_Node  *p=NULL; // Traversal pointer

  p=head;    // Initialize the traversal pointer to
          // point to the head node
  while (p!=NULL)
  {
        // Print out the review at this node
     printf("**********************************************\n");
     printf("Restaurant Name: %s\n",p->rev.restaurant_name);
     printf("Restaurant Address: %s\n",p->rev.restaurant_address);
     printf("Restaurant Score: %d\n",p->rev.score);
     printf("**********************************************\n");
     // Update the traversal pointer to point to the next node
     p=p->next;
  }
}

Review_Node *delete_list(Review_Node *head)
{
    /*
        This function releases all memory reserved for restaurant
        reviews in our linked list. It returns NULL so we can
        update the head pointer in main() to indicate the list
        is empty
    */

  Review_Node *p=NULL;
  Review_Node *q=NULL;

  p=head;
  while (p!=NULL)
  {
       q=p->next;
       free(p);
       p=q;
  }

  return NULL;
}

Review_Node *search_by_name(Review_Node *head,\
                          const char name_key[MAX_STRING_LENGTH])
```

```
{
  // Look through the linked list to find a node that contains a
  // review for a restaurant whose name matches the 'name_key'
  // If found, return a pointer to the node with the review. Else
  // return NULL.

  Review_Node  *p=NULL; // Traversal pointer

  p=head;
  while (p!=NULL)
  {
    if (strcmp(p->rev.restaurant_name,name_key)==0)
    {
        // Found the key!
     return p;
        }
     p=p->next;
  }
  return NULL; // The search key was not found!
}

Review_Node *delete_by_name(Review_Node *head, const char name_key[])
{
  // This function removes the node from the link list that contains the
  // review with a matching restaurant name.

  Review_Node *tr=NULL;
  Review_Node *pre=NULL;

  if (head==NULL) return NULL; // Empty linked list!

  // Set up the predecessor and traversal pointers to point to the first
  // two nodes in the list.
  pre=head;
  tr=head->next;

  // Check if we have to remove the head node
  if (strcmp(head->rev.restaurant_name, name_key)==0)
  {
      free(pre);  // Delete the first node in the list
      return tr;  // Return pointer to the second node (new head!)
  }

  while(tr!=NULL)
  {
      if (strcmp(tr->rev.restaurant_name, name_key)==0)
      {
              // Found the node we want to delete
          pre->next=tr->next;     // Update predecessor pointer
          free(tr);      // remove node
        break;           // Done!
      }
      tr=tr->next;
      pre=pre->next;
  }
  return head;   // Head did not change
}

int main()
{
```

```
Review_Node *head=NULL;
Review_Node *one_review=NULL;
char name[MAX_STRING_LENGTH];
char address[MAX_STRING_LENGTH];
int score;
int choice=1;

while (choice!=5)
{
  printf("Please choose one of the following:\n");
  printf("1 - Add a new review\n");
  printf("2 - Print existing reviews\n");
  printf("3 - Update review for one restaurant\n");
  printf("4 - Delete a review\n");
  printf("5 - Exit this program\n");

  scanf("%d",&choice);
  getchar();

  if (choice==1)
  {
     // Get a new review node
     one_review=new_Review_Node();
     if (one_review==NULL)
     {
             printf("main(): Error! can not reserve space for a new node. Ending the program
                 now\n");
             return 1;
     }

     // Read information from the terminal to fill-in this review
     printf("Please enter the restaurant's name\n");
     fgets(name, MAX_STRING_LENGTH, stdin);
     printf("Please enter the restaurant's address\n");
     fgets(address, MAX_STRING_LENGTH, stdin);
     printf("Please enter the restaurant's score\n");
     scanf("%d",&score);
     getchar();

     // Fill-in the data in the new review node
     strcpy(one_review->rev.restaurant_name,name);
     strcpy(one_review->rev.restaurant_address,address);
     one_review->rev.score=score;

     // Insert the new review into the linked list
     head=insert_at_head(head,one_review);
  }
  else if (choice==2)
  {
     print_reviews(head);
  }
  else if (choice==3)
  {
     printf("Which restaurant's score do you want to update?\n");
     fgets(name,MAX_STRING_LENGTH,stdin);
     one_review=search_by_name(head,name);
     if (one_review==NULL)
     {
        printf("Sorry, that restaurant doesn't seem to be in the list\n");
     }
```

```
      else
      {
         printf("Please enter the new score for the restaurant\n");
         scanf("%d",&one_review->rev.score);
         getchar();
      }
   }
   else if (choice==4)
   {
           printf("Which restaurant's review do you want to delete?\n");
           fgets(name,MAX_STRING_LENGTH,stdin);
           head=delete_by_name(head,name);
   }
 }

 // User chose #5 - Release memory and exit the program.
 head=delete_list(head);
 return 0;
}
```

3.15 Building Programs that Work - Part 3

At this point, we are building programs that consist of many different (non-trivial) functions, they use arrays, pointers, strings, and can manipulate a significant amount of information.

We have to develop a process that will enable us to **check our programs for correctness** to the degree that we can be **highly confident** our software does not contain errors of logic or programming bugs, that it stores and manipulates data in the way that was intended by the algorithm we set out to implement, and that it correctly solves the problem or performs the task it was designed for **on any reasonable input** that may be presented to it.

Achieving the above requires **discipline** and the developing both **a habit for testing thoroughly** and **the skill for knowing when you have tested enough**. The latter only comes from experience, and you will develop it over time. But the **habit** can be developed with a bit of help from having a good software development process to follow.

In the previous Chapter, we set out a starting point for our software development process. We discussed the importance of **thinking through the problem carefully**, and **writing a detailed solution on paper** before any actual coding happens.

Here, we will add a **key component** of our software development process: **careful and thorough testing while we are developing a program**, and **thorough testing of the finished software**.

3.15.1 Testing software as we are writing it

The first type of testing that we have to perform **for every piece of code that we produce** has the job of checking **individual functions** or **self-contained sections** of our code for correctness. Here's a solid, basic process we can use to help us produce code that is correct as we are developing a complex piece of software.

In order to be able to properly test code that you are writing:

- **You must have completed the design** of the software you are writing - in particular, you need to know **which functions** will be required, and the order in which they are going to be used.

- For each function you have to implement, you need to know **what the input is** (data type(s), constraints on possible valid or invalid values, whether or not it uses arrays or pointers, etc).
- For each function you have to implement, you need to know **what is the correct output or result** after calling the function given some **specific input(s)** - you **can not test code if you do not know what the correct result of a test case should be**.

These conditions should all be satisfied if you are following the procedure we discussed in Chapter 2 for developing your solution.

How to test your code as you're writing it:

Step 1 Decide **the order in which you will implement the different functions** that are part of your design. This has to be done carefully, because you should **only use parts of the code that have already been tested** in building more complex functionality. For example, if we have two functions

```
int function_A(int n)
{
    // This function does some processing
    // on an input integer n, and returns
    // an integer value.
    // It doesn't call any other functions
    // in the program

    // Here we would have some code
    // that does something interesting

    return result; // And eventually a result is returned
}

int function_B(int n)
{
    // This function does some more complex
    // processing of an input integer n, and
    // returns an integer result.
    // But, at some point, it uses function_A()
    // as part of the process

    // Here we would have some code
    // that does part of the work,
    // then we call function_A() to do
    // something as part of the process

    r=function_A(x);

    // and then the processing continues...

    return result; // Until the result is returned
}
```

the point of the example above is not the actual instructions in either **function_A()** or **function_B()**, but instead, it is to show that in order to be able to **test that** *function_B()* **works correctly**, we **need to be sure that** *function_A()* **works correctly**. If we have not tested **function_A()** and made sure it works as intended, then we will not be able to tell if any problems found while testing **function_B()** are the result of a problem with its own code, or whether perhaps the problem is in **function_A()** instead.

As another example, in the program that implements the linked list of restaurant reviews, we have to **implement and test** the **new_Review_Node()** function before we can use it in order to create and then insert nodes into the linked list. We can not test **insert_at_head()** if we are not sure that nodes being allocated for us by the **new_Review_Node()** function are correctly structured and initialized.

This means that our program will be implemented and tested starting from simpler (self-contained) functions, progressing toward more complex functionality only when the required smaller pieces **have already been tested and found to be correct**.

Step 2) Create a function in the program that will serve as a **test driver**. This function will contain **all the different tests** that you will be running over the functions and parts of your code, along with (if appropriate) **the code that verifies the results of each test**. We will be adding tests to the **test driver** as we implement functions in the code.

Step 3) Repeat the steps below for each function you implement (in the order determined in **Step 1)**, until all the functions in your program have been implemented.

- Write down (on paper!) a **set of tests** designed to put this function through its paces, and see how it behaves under different conditions. For instance:
  - For functions that process input and produce a result, this means checking various **valid** and also **not-valid** inputs, and verifying the function **produces the correct result** for the valid cases, and behaves in a **reasonable way** for non-valid inputs (e.g. prints an error message, or returns a default value, but does not cause the program to crash or produce a result that looks valid).
  - For functions that work on arrays or strings, this means creating input cases that check the function works on **a wide range of possible inputs**. This means testing inputs of **different lengths, including empty ones**, as well as testing for **any special cases** such as strings that contain special characters, or are not properly terminated with an **end-of-string delimiter** (once more, the function should handle these cases gracefully, without causing the program to crash).
  - For functions that work on collections and use data structures (such as our linked lists), this will require you to write **sequences of operations** that have been carefully designed to test that the function handles the data correctly, preserves the correct organization of the data structure, and produces the correct result on the data and/or structure of the collection.
- Think of ways in which you may **break the function** (cause it to fail or produce the wrong result), this could include non-valid inputs, tricky cases, unique arrangements of items in a collection, etc. Turn these into separate tests.
- For each of the tests you came up with, write down (on paper!) **the expected (correct) result**. This could be a number, the content of an array or a string after the function is called, or the structure of a collection after using a function that modifies it. For tests intended to **break the function**, the expected result would be that the function **does not fail and behaves in a way that makes sense** even for weird, unexpected, or tricky inputs.
- Now add each **test** to the **test driver function** along with **code that checks that the result matches what you expected:**

  - The **test** usually requires you to **set up some data** (whatever is appropriate for the function to work with), and then **call the function** to work on your test case and produce a result.
  - The **correctness check** can be done in multiple ways. You could **hard code** the expected values in the **test driver** (e.g. by setting some variable, or array, or string to have the content you expect to have if the function worked correctly) and then check the function's output against these hard coded value(s). You could **write a function** that processes the result of the test to check it is correct (we did that in Chapter 2 to check the prime-finding function). You could check by hand (against your written test results) by having the **test driver** print out the relevant information .
  - However you choose to check, the **test driver** should **not continue past any test that failed**. That means that if at some point the expected result, and what your function did are not in agreement, the **test driver** has to **let you know which test failed** and exit so you can get to work on finding and correcting the problem (more on that in a moment).
- Run the **test driver**. If **your function passes all the tests you designed** you can move on to implementing (and testing) the next function in your design. If any of the tests failed, it's time to do some debugging!

This may seem like **a lot of work**, and it actually takes a good amount of thought, effort, and time. But **it reduces by a huge amount the time, thought, work, and frustration** that you will undergo while chasing bugs in code that was not tested as it was being developed. More importantly, **it leads to overall better software** that has **fewer bugs**, is **more reliable**, and **requires less maintenance**. So work hard to **develop the habit of testing as you write your programs**.

3.15.2 Testing the finished software

Testing every single individual function for correctness as we write our code **does not guarantee that the finished program will work as intended**. After we have completed the development process, it is **essential to test the completed program** and ensure that everything works well together. This could involve testing **independent subsets** of the software as well as the complete program.

Testing the completed software **requires you to think in terms of the user** or **the problem that the software needs to solve**, and to **come up with a thorough sequence of tests** that covers as many **reasonable use cases** for the software as you can think of. If you can interact with the user, you definitely want their help in designing a full set of tests for the completed program. If you don't then it's up to you to come up with the tests.

However you go about generating the tests, it's important to keep in mind that:

- The tests must cover **all the common, typical operations** the user may perform on the software.
- The tests must consider a **realistic amount of data**. Many common problems show themselves only after a certain amount of data is being manipulated.
- They must provide a **wide variety of reasonable inputs** that achieve meaningful coverage of normal software use. This will involve **different input lengths**, **different sequences of operations**, and/or **different paths to completion** for the task the program is performing.
- The tests must also include **simulating input or behaviour** that may be expected from users with different levels of familiarity with the software.

- For programs that handle **sensitive information**, the tests must include **checking that information is properly protected** throughout the entire process, and no sensitive data is **exposed to unauthorized users**.
- In this final case, testing should also **simulate input or behaviour** that may be expected from a malicious user attempting to gain access to sensitive information.

You can follow a process similar to the one described above for the testing on individual functions, except this time it applies to the finished program. You may need to implement a separate test driver function to carry out these tests, as they will likely be more involved, require more input data to be set up, and require more work to check for correctness.

Ultimately **there is no single set of steps** to be followed in order to design and carry out the testing for any given program. Testing must reflect the complexity of the software we are developing, the type of information it deals with, the requirements set forth by the user (or in the absence of a user, the requirements set by the problem being solved through the software). You have to decide for each program, within its own context, how much testing is needed at each level to convince yourself that the software performs its work correctly and reliably.

Much like in Physics, **we can never categorically state that a complicated piece of software is bug free**. Through careful testing we can ensure that we have removed as many bugs as possible, and that the likelihood that a serious issue remains in the code is small.

Software testing is a complex, rich, and interesting area of study and research. What we have described above is **not a formal testing methodology**, it is just a starting point for good software development practice. We will continue to develop our ability to test and verify software for correctness as we gain experience and expand our knowledge of computer science and its many sub fields.

Chapter 4 Solving Problems Efficiently

Up to this point, we have learned how to **represent**, **store**, **organize**, and **search** through a collection of possibly complex data items. We know what an **Abstract Data Type** is and why they are useful for coming up with solutions to problems in a way that makes the solution independent from a specific implementation. We studied **lists**, and their implementation as **linked lists**, and we used these for building a simple database able to answer a few queries on items in our collection.

We will no doubt use lists in the future for a variety of applications. So before we close our discussion on lists, it's worth taking time to think about **how efficient** they are as a solution to problems that require frequent look-up of items in the collection (whether it is to look at their content, update the data stored there, or carry out aggregate computations).

Suppose we wanted to use linked lists to implement a fully functional database engine. The database should be able to store information for a large number of data items (think big - tens of millions, hundreds of millions, or even billions of nodes in the linked list). The question is, **what can we expect in terms of the time it takes for us to perform a search on the database?**

Note

Having such a large number of data items in a collection is very common.

- IBM's Watson used millions of documents to answer game questions for Jeopardy
- Google Image Search was indexing well over 10 billion images by 2010
- Facebook had about 1.5 billion daily active users in September 2018
- Amazon sells over 3 billion different products worldwide

So, thinking about what is the most efficient way to maintain a very large collection is of great importance.

To fully understand this question, we have to think a bit about **how data was stored in the list**. A common assumption, and one that is reasonable for many database applications, is that the **data was added in no particular order**, or what is the same thing the **entries** in the list **are randomly ordered** with respect to any data fields we may want to use for answering database queries.

We also typically assume that **queries arrive in random order**. This is a reasonable assumption - think about Google searches arriving over a span of 1 minute from everywhere in the world - chances are, there will be no order or pattern to the searches carried out by people from Mexico, Australia, Singapore, and Finland, all of whom happen to be online at that particular time.

Finally, we have to provide some **definition of efficiency** so we can evaluate how well our list is doing. For a database engine, one reasonable measure is **how many data items have to be inspected** before we find the one we are looking for.

Now, recall from the last unit that **to perform a search** in a linked list, **we have to do a list traversal**. This means starting at the head of the list, and traveling along the list's nodes until we find the one that contains the information we are looking for.

Question: How many nodes do we need to look at before we find the one we want?

- If we are **super lucky**, the data we want will be in the head node and thus **we only look at 1 node** to find what we want, but with randomly ordered data, the chance of this happening **is 1/N**, where **N** is the **number of nodes** in our list. For a list with 1,000,000,000 entries, the odds are very large against this ever happening.
- If we are **super unlucky**, the data we want will be at the tail of the list (or not in the list at all), and we have to look at **all N nodes** before we find it or can be sure it's not there.
- Most of the time, **we are neither super lucky, not super un-lucky**. The data is somewhere in the list, sometimes closer to the head, sometimes closer to the tail. Because the data is randomly ordered, the number of nodes we need to examine **averages out to N/2** after we run many, many queries.

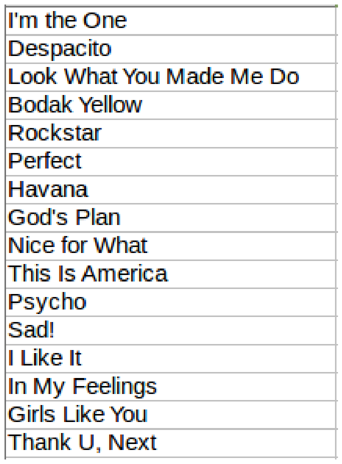
This makes sense: Sometimes we find the data closer to the head, sometimes closer to the tail, but these two balance each other out so on average we have to look at about half of the linked list to answer any given query.

This is really not too bad if our list is a few thousand entries long. But once our linked list grows into the millions of items and beyond, the cost of answering a query becomes too large. Simply put, our linked list has to look at too many data items in order to find a specific one. This translates into a longer wait time for a program requesting information from the database. At some point, the list is so long that the wait time becomes impractically long.

4.0.1 Why we need to look through all that data to find something

We may be inclined to think that the root cause of our problem is the structure of our linked list. And indeed, our **linked list** has a major limitation in that **we can not access an element inside the list without doing list traversal**. However, the root cause of our problem is not due to this limitation of the linked list.

Consider for example **an array** that contains the names of all of the hit songs from 2017 and 2018 as shown in Fig. 4.1 (it's a small list so we can show an example here, but you should be thinking this applies to an array of any size).



I'm the One
Despacito
Look What You Made Me Do
Bodak Yellow
Rockstar
Perfect
Havana
God's Plan
Nice for What
This Is America
Psycho
Sad!
I Like It
In My Feelings
Girls Like You
Thank U, Next

Figure 4.1: A **simple array** that contains the names of hit songs from 2017 and 2018.

The question we want to answer is: Is searching for a specific item in an array with randomly ordered entries more efficient than searching in a linked list with the same (unordered) entries? Or, what amounts to the same thing, we are asking if using an array allows us to find what we want with fewer than **N/2** items examined on average.

Looking at the array in 4.1, it should be fairly clear that **there is no pattern to how the data is ordered**. If we need to find a specific song name, we have to start at the top and look through the array until we find what we want. This is called **linear search**. Just like a linked list, there is a chance we get super lucky and our query is right at the first entry in the array, we can be super un-lucky so the query item is at the end, or not in the array; and on average we have to look through half the array before we find our information. If the array has **N** entries, we're back to looking through **N/2 entries on average**.

This means that **from the point of view of the number of items we need to look at** to answer a query, **a linked list and an array are equally efficient**. While there may be a slight performance difference in terms of actual run-time (because there is more overhead to a linked list than an array), **any small difference becomes irrelevant as the collection size grows**: both of these containers for storing information become inadequate for quickly answering queries.

4.0.2 Organizing our data for efficient search

If we are to find a faster way to answer queries, **we need to deal with the random order of our data**. We need to **sort it**. Once our data is in sorted order, we can do a much more efficient job of searching for specific items. Fig. 4.2 shows a sorted version of the array from Fig. 4.1.

Bodak Yellow
Despacito
Girls Like You
God's Plan
Havana
I Like It
I'm the One
In My Feelings
Look What You Made Me Do
Nice for What
Perfect
Psycho
Rockstar
Sad!
Thank U, Next
This Is America

Figure 4.2: A **sorted array** that contains the names of hit songs from 2017 and 2018.

For a person, **looking up a song in the sorted array above is much easier**. We understand the entries are in order, so we can easily find the spot where a particular song should be. This is why all content indexes in the backs of books, library shelves, or the phone directory (back in the days when it was actually a printed book) are sorted.

In programming terms, **sorting data has the immediate benefit of making that data much easier to search**, with the result that on a sorted array of pretty much any size, we only have to examine a few items in order to find what we're looking for.

4.0.3 Binary Search

The process of finding an item in a sorted array is called **binary search**. Suppose we want to find the song **"I'm the One"** in the array above. The binary search process is illustrated in Fig. 4.3.

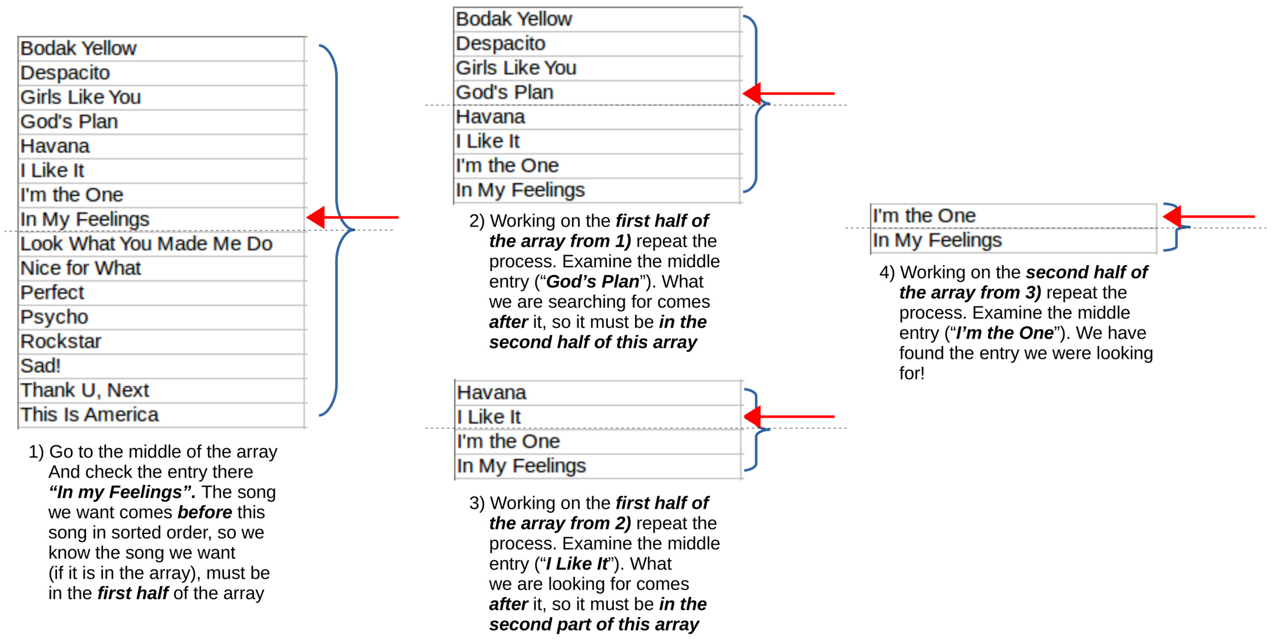

Bodak Yellow
Despacito
Girls Like You
God's Plan
Havana
I Like It
I'm the One
In My Feelings
Look What You Made Me Do
Nice for What
Perfect
Psycho
Rockstar
Sad!
Thank U, Next
This Is America
1) Go to the middle of the array And check the entry there *"In my Feelings".* The song we want comes *before* this song in sorted order, so we know the song we want (if it is in the array), must be in the *first half* of the array
2) Working on the *first half of the array from 1)* repeat the process. Examine the middle entry ("*God's Plan*"). What we are searching for comes *after* it, so it must be *in the second half of this array*
3) Working on the *first half of the array from 2)* repeat the process. Examine the middle entry ("*I Like It*"). What we are looking for comes *after* it, so it must be *in the second part of this array*
4) Working on the *second half of the array from 3)* repeat the process. Examine the middle entry ("*I'm the One*"). We have found the entry we were looking for!

Figure 4.3: The **binary search** process illustrated on a small sorted array.

Binary Search Algorithm

- 1) Go to the entry at the **middle of the current array** - this is the entry at index $\lfloor N/2 \rfloor$ (the **floor of N/2**).
- 2) **Compare** the middle entry with the **query term**, if they match **return the index of the matching item**.
- 3) If the **current array** has only one entry, the **query term** is **not in the array**, return **-1** to indicate the term was not found.
- 4) If the **query term** is **less than** the **middle entry in the current array**, then it has to be **in the first half of the array**, so **the first half becomes the current array, go back to 1)**. Otherwise, the **query term** must be in the **second half of the array**, so **the second half becomes the current array, go back to 1)**.

Binary search is a straightforward process, and it is **incredibly efficient**. This is not evident in the example from Fig. 4.3 because the array shown there is tiny. Let's think about what happens with larger arrays.

Key ideas:

- The **binary search** process **uses the fact that the array is sorted** to **predict** in which half of the array the data we want has to be.
- This means that **for every item we check, we can discard from the search half of the remaining entries**. When we choose which half of the array to search next, we can be sure we will never have to look at any items in the other half.

- Thus, every round of binary search that we complete, the number of items that remain to be checked is cut in half.

This very quickly reduces large collections to very manageable sizes. For example, if our initial array has a length of **1024** entries

- After the first round of binary search, we are left with **512** entries to check
- After the second round of binary search, we have **256** entries left to check
- After round 3, we have **128** entries left to check
- After round 4, we have **64** entries left
- After round 5, we have **32** entries left
- After round 6, we have **16** entries left
- After round 7, we have **8** entries left
- After round 8, we have **4** entries left
- After round 9, we have **2** entries left
- After round 10, we have only **1** entry left to check

This is quite remarkable: In an array with over one thousand entries, we can find any item we want with **10 or fewer** checks. Compare that with the **un-sorted array** or the **linked-list**, in which case we would expect to have to check on average **512 entries**.

So **binary search** is **a lot more efficient** than **linear search** for finding information in large collections, but **just how efficient is binary search?**

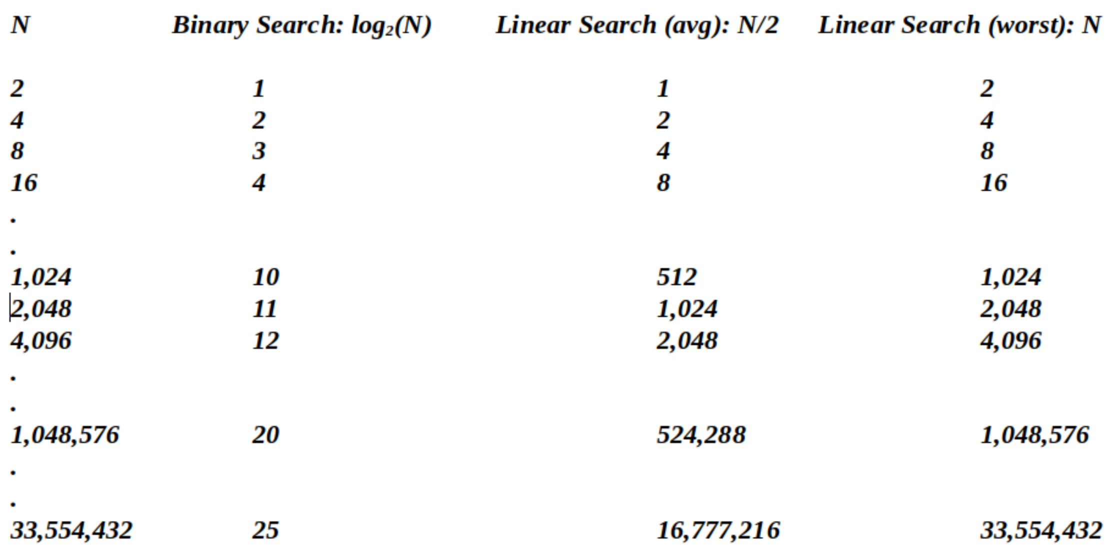
This is equivalent to asking: Given an initial value for **N**, the **number of entries in the array**, how many steps are needed to reach the point where only one element is left? (this tells us the maximum number of entries we need to examine to find what we want).

The answer turns out to be $k = log_2(N)$.

For the array with **1024** entries, $k = log_2(N) = 10$, which is exactly the number of steps we had to do above to get to a single item. To get a sense of just how big a difference this makes, Fig. 4.4 shows a comparison of the number of checks that are performed by **binary search** compared with **linear search** as the size of the collection grows, and showing three cases:

- a) The maximum **(worst case)** number of items we need to look at **for binary search**
- b) The **average number** of items we need to look at **for linear search**
- c) The maximum **(worst case)** number of items we need to look at **for linear search**

From Fig. 4.4 it should be very clear to you that **binary search is incredibly efficient**. Even in the worst possible case, with only **25 checks** we can find any one item in a collection with **over 33 million entries**! Conversely, **linear search** on an un-sorted array or linked list would be expected to have to check **over 16 million items**, on average, and all **33 million** if we are unlucky.



N	Binary Search: $log_2(N)$	Linear Search (avg): N/2	Linear Search (worst): N
2	1	1	2
4	2	2	4
8	3	4	8
16	4	8	16
.			
.			
1,024	10	512	1,024
2,048	11	1,024	2,048
4,096	12	2,048	4,096
.			
.			
1,048,576	20	524,288	1,048,576
.			
.			
33,554,432	25	16,777,216	33,554,432

Figure 4.4: Comparison of the amount of work done by **binary search** (worst case) and **linear search** (on average, and worst case) as the size of the collection grows.

4.1 Computational Complexity

The ideas above can be refined into a **concrete, general framework** for comparing different algorithms in terms of the **computational cost** of carrying out a task.

One key idea in understanding how we can compare different algorithms is that we want to find **a measure of the amount of work a particular algorithm has to do as a function of N**, the number of data items the algorithm is working on. We call this measure the **computational complexity of the algorithm**.

In the examples above, we saw that linear search has to examine **N** items in the worst case, while binary search only has to look at $log_2(N)$. The reason for wanting to look at a function of **N** is that **it allows us to predict how an algorithm will perform for increasingly larger data collections**. In our search example, we have two different functions:

$$f_{BinSearch}(N) = log_2(N)$$
$$f_{LinSearch}(N) = N$$

So we say that **binary search has a complexity of** $log_2(N)$ and that **linear search has a complexity of N**. Because the value of $log_2(N)$ grows much more slowly than **N**, we can conclude (without having to test on every possible array) that **binary search is much more efficient than linear search for large collections**. We can visualize this by plotting both functions for increasing values of **N** and comparing the **predicted** amount of work they will have to do, this is shown in Fig. 4.5.

It should be obvious from the plots in Fig. 4.5 that one of these algorithms is a whole lot more efficient than the other at finding information in a large collection. The key observation we can make from this is:

Given two algorithm and the functions of **N** that describe their computational complexity, the **function that grows more slowly will always win-out as the number of data items increases**. So the algorithm with the

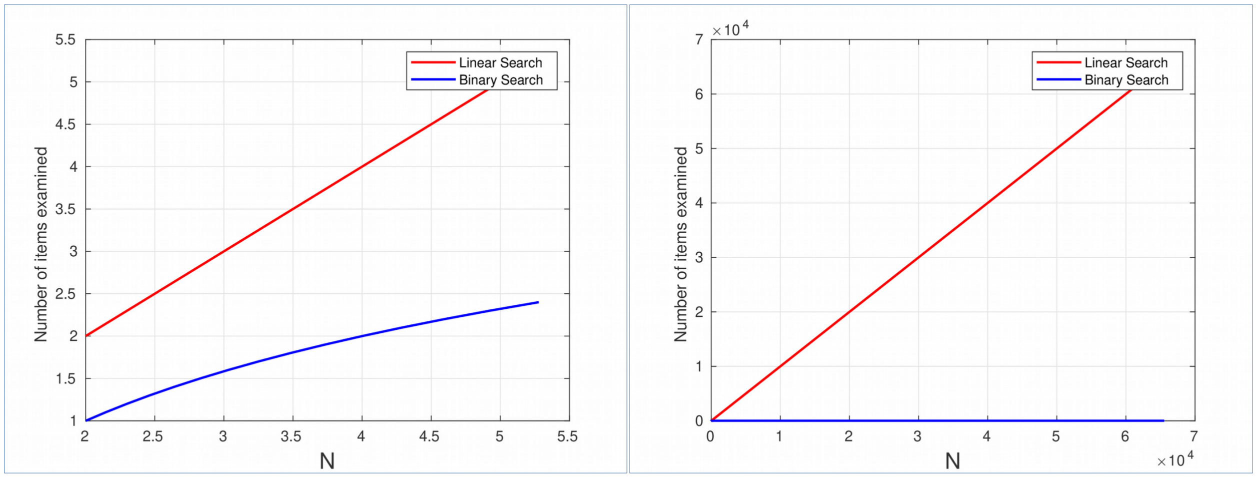


Figure 4.5: Visualization of the **complexity of two different search algorithms**: **Linear search** (red) and **binary search** (blue). Binary search is incredibly efficient, to the point of looking like it does no work at all compared to linear search on large arrays. The shape of each function's curve is easier to compare on a graph for smaller values of N (left side graph)

slower-growing function is said to have a **lower complexity**, and is therefore shown to be **more efficient** for large data collections.

Question: What can we conclude at this point about the run-time of two programs that do search on the same data, but where one uses binary search, and the other uses linear search?

Being able to compare different possible ways of carrying out some task is essential for implementing good solutions to any problem. In computer science, **we compare algorithms by studying and measuring their computational complexity**. Computational complexity is expressed as a function of **N**, the number of data items the program has to work on.

4.1.1 The Big O Notation

In order to be able to better understand and compare the complexity of different algorithms, we use a special notation called the **Big O** notation. The importance of the **Big O** notation is that it allows us to reason about the efficiency of algorithms in terms of general classes of functions. Here's what we mean by that:

Suppose that we have **different implementations of linear search** on arrays (perhaps written by different developers, in different programming languages), and we also have **different implementations of binary search**. We then set out to **carefully measure their actual run-time** for arrays of different sizes, on a particular computer, and we find that they behave as follows:

- a) $.75 \cdot N$ (e.g. for N=10, this implementation runs in 7.5 seconds)
- b) $1.25 \cdot N + .25$ (this one has a .25 second start-up time that is not dependent on **N**)
- c) $2.43 \cdot N$
- d) $1.15 \cdot log_2(N) + 1.12$ (this one also has a start-up time, in this case 1.12 seconds independent of **N**)

- e) $1.75 \cdot log_2(N)$
- c) $15.245 \cdot log_2(N) + 15.25$ (large start-up time! maybe it needs to load libraries before it runs)

The results above require some thought - we know how much work linear search has to do to find an item in the array, and that in the worst case that would be looking at all **N** entries in the array. But **we hadn't considered what could happen when variations between implementations are introduced**.

Perhaps an implementation in **C** will be slightly faster than one in **Java**, and perhaps the one in **Java** will be somewhat faster than one written in **Matlab**. But the important thing to note is that **all of them are linear functions of N**. It is a **property of the linear search algorithm**. The only thing that can change among (correct) implementations is the **constant that multiplies N**, and maybe the presence of a **constant term independent of N**.

We can say exactly the same in a different way: Any correct implementation of the linear search algorithm will have a complexity that is characterized by $c \cdot N + k$, where **c** and **k** are constants, for large values of **N**.

The same is true for different implementations of binary search. They are all **logarithmic functions of N** multiplied by some constant that is implementation dependent, and perhaps including a constant term independent of N.

We want a way to compare algorithms in an implementation independent way. We need to know which is the better algorithm, the one requiring less work to solve a given task. And if we can analyze algorithms in this way, we can always pick the most efficient one. The **Big O** notation makes explicit the **slowest-growing** function of **N** that characterizes the **complexity of some algorithm**. The definition of **Big O** states that:

Definition 4.1 (Big O Notation)

To state that $f(N) = O(g(N))$, which is to say that **a function** *$f(N)$ has a* **Big O** *complexity of $g(N)$ means that $f(N) \leq c \cdot g(N)$ for a sufficiently large value of N (usually stated as $N > N_0$). It is assumed that $f(n)$, c, and $g(n)$ are positive.* ♣

As an example, $f(N)$ could represent the run-time of **binary search**. We know that a correctly implemented binary search will have to check **at most** $log_2(N)$ entries to find the one we need, so we can state $f(N) = O(log_2(N))$ - this immediately tells us that we expect $f(N) \leq c \cdot log_2(N)$ for sufficiently large **N**. We are categorically stating that **whatever the implementation of binary search** (regardless of programming language, which computer it's running on, or who wrote the program), we can **always find a function** $g(N) = c \cdot log_2(N)$ such that the run-time of the algorithm $f(N)$ is always less than, or equal to $g(N)$.

We say that **binary search has a worst-case complexity of** $O(log_2(N))$. Similarly, we can state that **the worst-case complexity of linear search is** $O(N)$ because whatever the implementation, we can always find a function $g(N) = c \cdot N$ such that the runtime $f(N)$ of linear search is always less than or equal to $g(N)$.

From the above, we can categorically conclude that **binary search** is significantly more efficient than **linear search** for large **N**, and this is regardless of any possible differences in the implementation of the algorithms, or the hardware they are running on. We can visualize why this statement is true by looking at the **run-time plots** for the different implementations of **linear search** and **binary search** listed above. This is shown in Fig. 4.6.

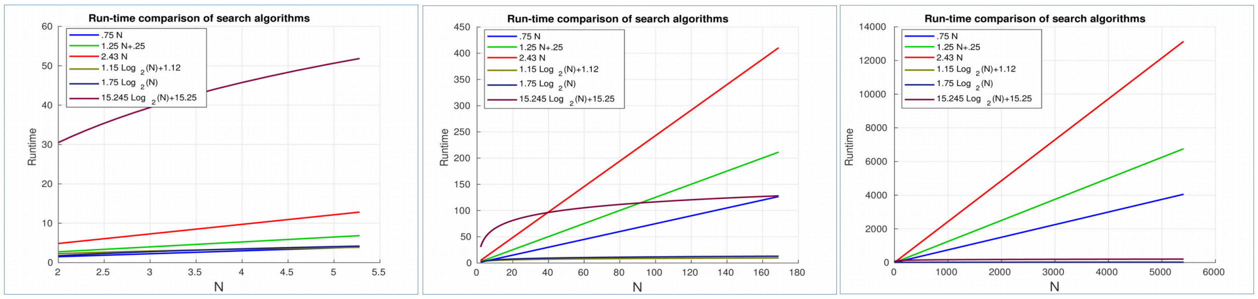


Figure 4.6: Plots of **run-time** for the different versions of **linear search** and **binary search**. Each graph shows what happens for different ranges of values of **N**, increasing from left to right. For small values of **N**, linear search would appear to be more efficient. However, by the time **N=170** all the linear search implementations have overtaken the slowest binary search implementation, and by the time **N=1000** it's **all over for linear search**. The conclusion is: **Binary search will win out in the end, regardless of how it's implemented, if N is large enough.**

The point not to be missed in the plots shown in Fig. 4.6 is that **the algorithm with the smallest Big O complexity will win for large enough N**. The $log_2(N)$ function (or in fact, **any log function, regardless of the base**) is **much slower growing than any linear function**, so we will always expect binary search to be faster on a large enough array. The note about any log function being slower growing than any linear function means we usually **ignore the base of the** log **function**, and we simply state that **binary search is** $O(log(N))$.

Note

Why do we want the slowest-growing function that characterizes the complexity of an algorithm? Consider the graph shown in Fig. 4.7 that plots the runtime of an implementation of **binary search** written in **C** for various values of **N**.
The plot also shows two functions both of which can be used as **worst case upper bounds** on the algorithm's run-time. From the definition of **Big O** complexity, if we take $f(N)$ to be the run-time for the binary search program, both of the following statements are correct:

- $f(N) = O(log_2(N))$
- $f(N) = O(N)$

The Figure clearly shows we can find a **logarithmic** function such that $c_1 \cdot log_2(N) \geq f(N)$ for all values of N in the plot. We can also find a **linear** function such that $c_2 \cdot N \geq f(N)$ for all N in the graph. However, notice that the **linear** function really does not provide a good **prediction** for what the run-time of the program will be for any particular value of N. Indeed, for large N, the **linear** function **over-estimates run-time by a very large amount**, whereas the **prediction from the logarithmic function is much closer** to the real-world behaviour of the program.

Therefore, in order to be able to reason about, or compare the amount of work that different algorithms will have to do while solving a problem, we need to find **the slowest-growing function of N** that provides a bound on how much work the algorithm does as **N** grows.

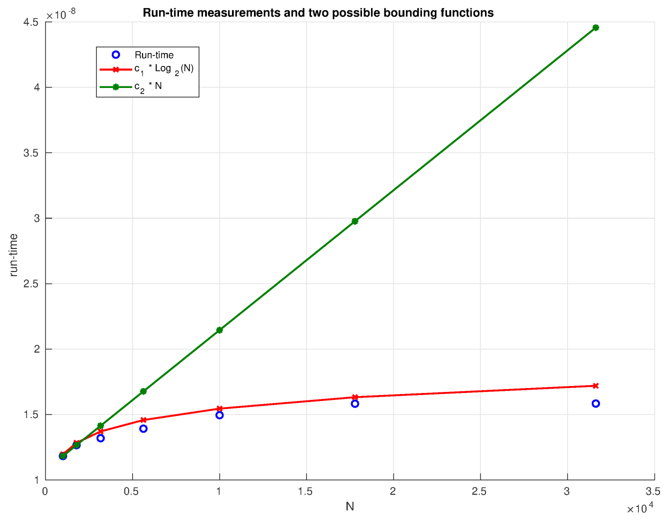


Figure 4.7: Run-times for a **C** implementation of **binary search** (blue markers) for different array sizes (x-axis). The two functions shown can both be used as bounds for the algorithm's run-time. However, the **linear** function is not useful for **predicting** what the run-time should be for a particular value of **N**.

4.1.2 From algorithm complexity to problem complexity

We started with the goal of **understanding how efficient** two different algorithms for finding a particular item in an array can be. Complexity analysis is a wonderful tool for doing this. However, there is an even more powerful idea behind the use of complexity analysis in computer science:

We can **study, quantify, and prove** results regarding the **complexity of solving a given problem**. What is the difference? Let's take an example: **Sorting an array** (and we will come back to this example soon in more detail).

- **Algorithm complexity** means we can look at different algorithms to sort an array, and figure out which one is going to be more efficient as the array size grows.
- **Problem complexity** means we study the actual problem of sorting, and we try to figure out what is the **theoretical lower-limit** on how much work the best possible algorithm has to do to sort an array.

For example, **proving that linear search has a complexity of** $O(N)$ is equivalent to showing that it is **not possible** to find an algorithm that can do linear search with complexity less than $O(N)$.

This is a powerful and important idea because it **allows us to classify problems** by **how hard** (in terms of complexity) they are to solve computationally. Harder problems are characterized by a **Big O** complexity given by fast-growing, or very-fast growing functions of **N**. As a developer, it's important to become familiar with the different classes of problems you may have to work with one day, and to consider carefully what is reasonable to expect of an algorithm, given the complexity of the problem it is working on. A comparison of a handful of important algorithms and problems in terms of their computational complexity is shown in Fig. 4.8.

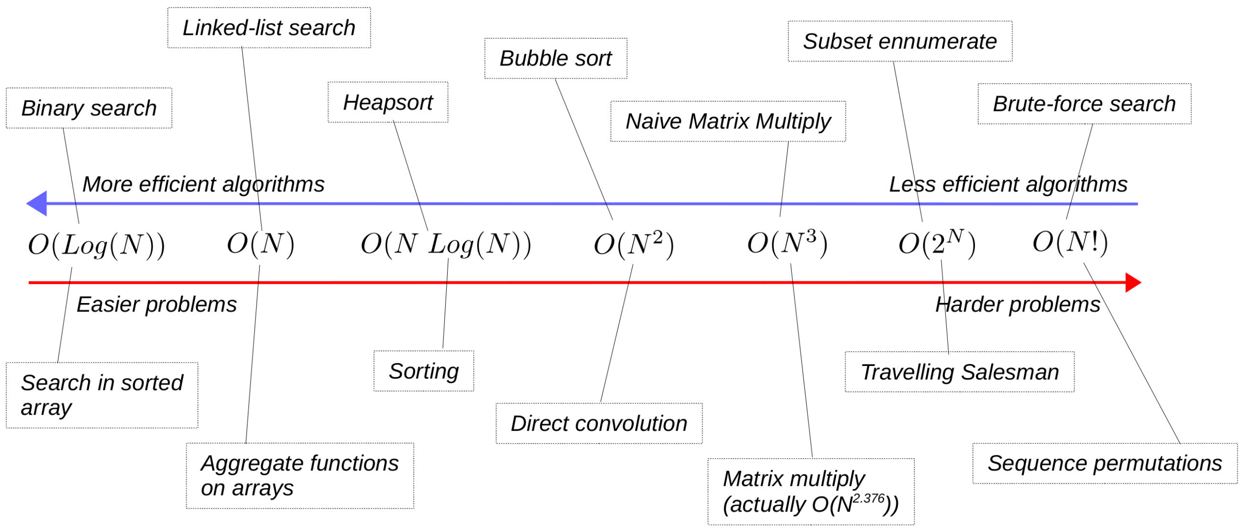


Figure 4.8: The plot above shows examples of **algorithms** (on top) and **problems** (at the bottom) with different **Big O** complexity. Note that for a given problem (e.g. sorting), you can easily find algorithms whose complexity is greater (bubble sort) than the best known complexity for the problem. Knowing what the best known complexity estimate is for a given problem allows you to evaluate how good an algorithm is at solving that problem.

Note

We can measure complexity in terms of different quantities: **the number of times we have to check an item** in a collection, **the run-time** of an algorithm, **the number of mathematical operations** a certain function has to perform. Which measure to use **depends on what problem we are solving**, but the analysis, and what it tells us about an algorithm or problem is stated in the same way using the **Big O** notation.
Computational complexity analysis applies to **every type of problem**, not only searching for information in a collection. For example, it is incredibly important in all areas of computer science that deal with numerical computation, for example machine learning and artificial intelligence. We can learn a lot more about computational complexity, and about how to study, analyze, and prove estimates of a problem's complexity by reading up on the topic. It requires a deeper and more serious study than is appropriate for this book.

4.1.3 So this means the implementation doesn't matter right?

Not quite - solving a problem efficiently requires us to **first think of the complexity of the algorithm** for solving the problem. Once we have selected an algorithm with the best known complexity for solving a particular problem, we have to write a good implementation of it.

Going back to our original example of search algorithms, the difference between the implementations of algorithms that have **the same Big O complexity** becomes important once **N** is large enough, e.g. **N=1000000**:

- $1.15 \cdot log_2(N) + 1.12 = 24.041$
- $15.245 \cdot log_2(N) + 15.25 = 319.11$

So the implementation makes all the difference between us having to wait around **20 sec.** for the result, or having to wait **over 5 minutes**!

✍ **Exercise 4.1** Consider a function that takes 2 input arrays:

```
  void multiply_accumulate(float input[N], float output[N])
```

The function computes each value in the **output** array as $output[i] = \sum_j input[i] \cdot input[j]$

That is, the i^{th} entry in the output array is the result of multiplying the i^{th} entry in the input array with every other entry in the input array (including itself), and adding all of these up.

One way to implement this function is shown below:

```
void multiply_accumulate(float input[N], float output[N])
{
  int i,j;

  for (i=0; i<N; i++)
  {
    output[i]=0;
    for (j=0; j<N; j++)
    {
      output[i]=output[i]+(input[i]*input[j]);
    }
  }
}
```

Question: What is the **Big O** complexity of the function shown above? For this exercise, define the complexity in terms of **how many multiplications are carried out**. If you're having a tough time figuring that out from the code, try and list the operations the loops are doing for a small value of **N** and see if that suggests something to you.

Question: Would the **Big O** complexity change if **instead of multiplications** we defined complexity in terms of the **number of additions** that are carried out by the function?

Question: Is the complexity result we found above the best that we can do for this problem? Is there a better algorithm? And if there is, what is the **Big O** complexity of that algorithm?

4.2 How to make search more efficient

We started this unit trying to understand how efficient linked lists are in terms of letting us search for items in our collection. We have now seen that searching in linked-lists has a **Big O** complexity of $O(N)$, and because of that, for large collections, they are not the optimal way of storing information that will need to be searched frequently.

As a side question: Why is it not possible to perform binary search on a linked list whose items are in sorted order?

We also saw that **binary search** has a complexity of $O(Log(N))$ which makes it incredibly efficient for finding information even for very large collections. So based on this we may conclude that we should give up on linked-lists and that we should simply stick to sorted arrays of items so we can have efficient search.

However, we left arrays behind when we realized that their limitations (fixed capacity, either insufficient for data, or wasteful of space) made them hardly a good solution for the management and storage of large collections of items whose quantity is not known in advance, and in cases where the size of the collection can change a lot over time.

So we have a serious problem now:

At this point it looks like we can have either

1) A **dynamic data structure** that allows us to organize and maintain a large collection of items without wasting space.

or

2) A sorted array with fixed capacity (and the associated known limitations), but where we can perform binary search to efficiently search for information.

What are we missing?

Suppose we decided that efficient search is more important to us than not wasting space, so we are willing to use an array large enough for our collection, so we can do binary search and find items quickly. Sounds like a good plan, except that we haven't accounted for the fact that **we expect items to be added to our collection in random order**. Binary search requires our array to be sorted.

So before we decide that our best bet is to use a sorted array along with binary search, **we have to consider the computational cost** (and you know now that by this we mean the computational complexity in terms of **Big O**) **of sorting** the array in the first place.

4.2.1 The computational cost of sorting

Consider an (unsorted) array with **N** entries. Let's consider the simplest sorting algorithm we can come up with: **bubble sort**. If you haven't seen bubble sort before, here's a simple implementation:

```
void BubbleSort(int array[], int N)
{
   // Traverse an array swapping any entries such that
   // array[i] < array[j]. Keep doing that until the
   // array is sorted (at most, N iterations of the
   // loop on i)

   int t;

   for (int i=0; i<N; i++)
   {
     for (int j=i+1; j<N; j++)
     {
       if (array[i]<array[j])
       {
         t=array[j];
         array[j]=array[i];
```

```
            array[i]=t;
         }
      }
   }
}
```

In **bubble-sort**, the larger elements **float to the top of the array** like bubbles. After the first iteration of the loop on **'i'**, the largest number in the array is in the correct location (the first entry in the array), after the second iteration of the loop on **'i'** the next largest number is at the correct location (second entry in the array), and so on.

The question is: what is the worst-case **Big O** complexity of **bubble sort**? In this case, let's measure work in terms of **how many comparisons** of pairs of entries the algorithm has to perform - which is a common measure of work for **sorting methods**. Let's take a look at how many swaps would be needed to sort the array:

- For the first iteration of the loop on **'i'**, the loop on **'j'** would perform **N-1** comparisons
- For the second iteration of the loop on **'i'**, the loop on **'j'** would perform **N-2** comparisons
- For the third iteration of the loop on **'i'**, the loop on **'j'** would perform **N-3** comparisons
- ... and by now you can see the pattern arising ...
- For the $(N-1)^{th}$ iteration of the loop on **'i'**, the loop on **'j'** would perform **1** comparison
- And finally, for the N^{th} iteration of the loop on **'i'**, the loop on **'j'** would perform **0** comparisons

The total amount of work performed by **bubble sort** is the sum of the comparisons performed for the N iterations of the loop on **'i'**. We can figure out what that is by noting that matching pairs of iterations combine to perform a constant number of comparisons. The **first** and **last** iteration combined carry out $N-1$ comparisons ($N-1$ from the first iteration, 0 from the last one). The **second** iteration and the **next-to-last** iteration combined also carry out $N-1$ comparisons ($N-2$ and 1, respectively). The **third** iteration and the **second from last** combine to carry out $N-1$ comparisons as well, and so on. In total, there are $N/2$ pairs of iterations each of which combines to perform $N-1$ comparisons. So the total number of comparisons performed by the **bubble sort** algorithm is:

$$\frac{N}{2}\cdot(N-1)=\frac{N\cdot(N-1)}{2}=\frac{N^2}{2}-\frac{N}{2}$$

Question: The total work done (in the worst case) by **bubble sort** has two different terms both of which involve N, so what is the worst-case **Big O** complexity of **bubble sort**?

To answer that question, all we have to do is consider what happens to each of these terms as N grows. The quadratic term $N^2/2$ grows **a whole lot faster** than the linear term $N/2$. For instance, for $N=1000$, we have that $N^2/2=500,000$ and $N/2=500$ so the linear term is 1000 **times smaller**. As N **grows to** 10000 we have that $N^2/2=50,000,000$ and $N/2=5000$ which is $10,000$ **times smaller**.

From this you can glean an important principle: Whenever you have a complexity result that involves **different terms involving N**, the **fastest growing term** will **dominate the complexity of the algorithm as N grows**.

In the case of **bubble sort**, that means we need only consider the **quadratic term** $N^2/2$. And we say that worst-case **Big O** complexity of **bubble-sort** is $O(N^2)$.

Note

Don't forget that this is appropriate for comparing and choosing among algorithms that have different **Big O** complexity. If what we are doing is **trying to choose between algorithms that have the same Big O complexity** (e.g. the different versions of binary search we discussed earlier on) then we have to include **everything**. That means, any terms that depend on N as well as the constants that may appear in our estimate of how much work the algorithms are doing.

Houston, we have a problem:

- We want to use a **sorted array** and **binary search** so that searching for items in our large collection has a complexity of $O(log(N))$
- But in order to do that, we have to first sort the array, which (if done with bubble sort) has a complexity of $O(N^2)$

that is not a nice result. The **quadratic cost of sorting** will **dominate** the complexity of the process, and this cost grows **a lot faster** than even the **linear search** complexity which is only $O(N)$. Suddenly, using a **sorted array** doesn't seem like such a good idea.

4.2.2 It gets more interesting

Bubble sort is definitely not the best sorting algorithm out there (President Obama knows this, see Fig. 4.9) - we sometimes use it for small arrays because it's so simple and quick to implement, but we can do much better. If you recall from our discussion above on the **computational complexity of problems**, the best known sorting algorithms have a complexity of $O(N \cdot log(N))$. There are several algorithms that can reach this level of performance, including a couple we will study in detail in the next Chapter. For now let's see how far we can get if we replace our costly bubble sort with the more efficient **merge sort**.

Figure 4.9: "I think the bubble sort would be the wrong way to go". *Photo: U.S. Federal Government – public domain*

How would this change our decision on whether or not **sorting an array** and then using **binary search** to find information efficiently is reasonable? What we have now is the following situation:

- We have to sort the collection, which has a **worst case Big O** complexity of $N \cdot log(N)$ using **merge sort**
- Thereafter we can find any item in the collection in $O(log(N))$ time

Unfortunately, $N \cdot log(N)$ is still bigger than N, so we could be led into thinking that doing **linear search** remains the best option. But it's actually more complicated than that.

Suppose we are working with a collection that doesn't change much - for example, suppose we are maintaining a database of all the streets in a particular city. New streets don't appear every day, and existing streets don't disappear very often either. Under these conditions **we could do the sorting once**, and once we have a sorted collection we can carry out **many, many searches efficiently** using **binary search**.

In this situation, whether we should use **sorting and binary search** or **simple linear search** in an unsorted linked list or array comes down to **how many searches are we going to do?**. The table below illustrates the total cost of **carrying out a sequence of searches** using either **sorting and binary search** or an **unsorted array**.

Example 4.1

```
In the table below, N is the number of entries in the collection, M is the
number of searches we want to perform, 'A' is the cost of
sorting the array (it's done only once), 'B' is the total cost of doing M
searches using binary search (M*log(N)), and 'C' is the cost of
linear search (N*M). The line marked [+] indicates where the cost of
using linear search (C) becomes greater than the cost of sorting and using
binary search (A+B).

N=1000

                  A                              B                          C
M          Sorting Cost (N * log(N))    Searching Cost (M*log(N))    A+B            N*M

1               6907.8                         6.9                6914.7        1000
10              6907.8                        69.0                6976.9       10000  [+]
100             6907.8                       690.7                7598.6      100000
1000            6907.8                      6907.8              138165.6     1000000

N=10000

                  A                              B                          C
M          Sorting Cost (N * log(N))    Searching Cost (M*log(N))    A+B            N*M

1               92103                          9.2               97112         10000
10              92103                         92.1               97195        100000  [+]
100             92103                        921.0               98024       1000000
1000            92103                       9210.3              106313      10000000
```

The table above is interesting because it shows that even though the initial cost of sorting can be significant (as seen, for instance, for $N = 10000$), that cost is the same whether we do 10, or 1000 searches. Because **binary search** is so efficient, it only takes a few searches to make **linear search** more costly than **sorting once and searching many times**. In both of the cases above, it takes only 10 searches for **linear search** to do more work than our solution using **merge sort** and **binary search**.

So, you should keep in mind that **Big O** complexity results have to be used carefully - the actual decision on **what algorithm or combination of algorithms** you should use for working with a particular collection involves not only the **Big O** complexities for each of the components of the algorithms you're comparing, but also **the type of operations** you are expecting will be carried out on the collection, and **their frequency** (how often they take place).

If our collection had frequent insertions and/or deletions, we would need to **re-sort** the collection (or use **insertion sort**) to keep the array in sorted order, and if this happens often that would quickly make the solution that uses sorting much worse than the **linear search**.

✍ **Exercise 4.2** Build a table like the one shown in Example 4.1, But this time, in addition to the searches, there are **Q** insertions, where **Q** can be **10** or **100** and after each insertion **the array must be re-sorted** (i.e. the cost of **each insertion** is $N \cdot log(N)$).

✍ **Exercise 4.3** Build a table like the one shown in Example 4.1, but instead of **merge sort**, we are using **bubble sort** for sorting the array before doing **binary search**. Fill in the table and find out **how many searches are needed** for **linear search** to become less efficient than the combination of **bubble sort** and **binary search**. Your table should show what happens for **N=1000** and **N=10000**.

So have we found an efficient way to make a large collection searchable?

Not quite...

- The array solution has space limitations (fixed size can be either insufficient, or wasteful of space).
- It requires sorting. With the best sorting algorithm this has complexity $O(NLog(N))$ which is already worse than linear search (it can still be worth it if we sort once and then search many times).
- But, we expect most collections to change over time - insertions, deletions, and modifications will require work to ensure the array remains sorted.
- The conclusion is that a **sorted array with binary search is not necessarily the best solution** for organizing, storing, and searching over a large and changing collection of items.

What do we need?

- It is clear that for efficient search **data has to stay organized in some way**. We need an **ADT** than can facilitate this without requiring a separate sorting step.
- The **ADT** must support efficient **search**, **insertion**, and **deletion**; and these operations must remain efficient (in terms of **Big O** complexity) as the collection grows in size.
- The implementation of this **ADT** as a **data structure** should be **dynamic** and request/use space only as needed by the items in our collection.

We will now learn about an **ADT** and associated data structure that can do just that. We will study its properties, learn how it handles searches, insertions, and deletions, figure out the **Big O** complexity of the typical operations it supports, and look at some of its applications.

4.3 Trees, Binary Trees, and Binary Search Trees

Recall that a linked-list is a data structure in which nodes contain one item from a collection, and one link to a successor node which is the next node in the list.

A **tree is a generalization** of this idea, **it consists of nodes**, each of which contains **one or more data items** from a collection, and **one of more links** to children nodes (the equivalent of the successor node, but now we can have many).

Trees are extremely common structures in computer science, used for a wide range of purposes.

A particularly common type of tree is the **binary tree**, which has the property that **each node has two child nodes**, the left child, and the right child.

The diagram in Fig. 4.10 illustrates a binary tree. Note the following:

- Each node has **two spaces for links** - one for the left child (shown with red arrows), and one for the right child (shown with blue arrows).
- Nodes may have zero, one, or two children.
- The node at the top of the tree is called the **root node**. Similarly to the head of a linked list, we manage a tree by keeping a pointer to the root node.
- Each level in the tree consists of **nodes that are at the same distance or depth** with regard to the **root node**. Each level contains **up to 2 times** the number of nodes of the previous level.
- Leaf nodes are nodes with **no children**.

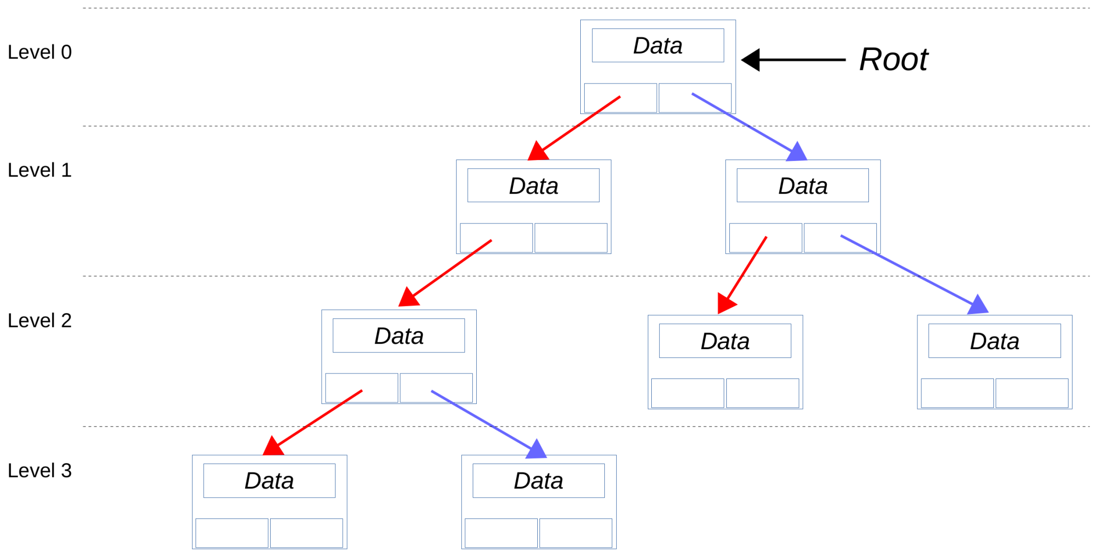


Figure 4.10: Structure of a **binary tree**, each node has up to **two children** and contains data for one item in a collection.

Without any additional refinements, binary trees may be useful for various tasks. However, for efficient management of a data collection, we have to satisfy the requirement that the tree keeps data organized in a way that makes the various operations efficient. With an additional constraint on how the tree is built we can achieve this last requirement.

4.4 Binary Search Trees

A **Binary Search Tree (BST)** is binary tree such that for each node in the tree, the **BST property** holds:

- Data in the **left sub-tree** of a node have value **less than, or equal to** the value of the data in the node
- Data in the **right sub-tree** of a node have value **greater than** the value of the data in the node

This means that at each node, we can quickly determine which of the following three situations is true:

- The data is in the current node
- The data is not in the current node, but if it is in the tree, it must be in the **left sub-tree**
- The data is not in the current node, but if it is in the tree, it must be in the **right sub-tree**

The above should make you think about binary search! The **BST** is intended to allow us to organize a large collection in such a way that we can quickly search through it. Indeed, under the assumption that our data is inserted into the tree in random order, the **BST** can provide an **average case** search complexity of $O(Log(N))$.

4.4.1 Search in a BST

The purpose of a **BST** is to support **efficient search** for data stored in the tree. Let's consider the search process in a **BST** as shown in the pseudocode below:

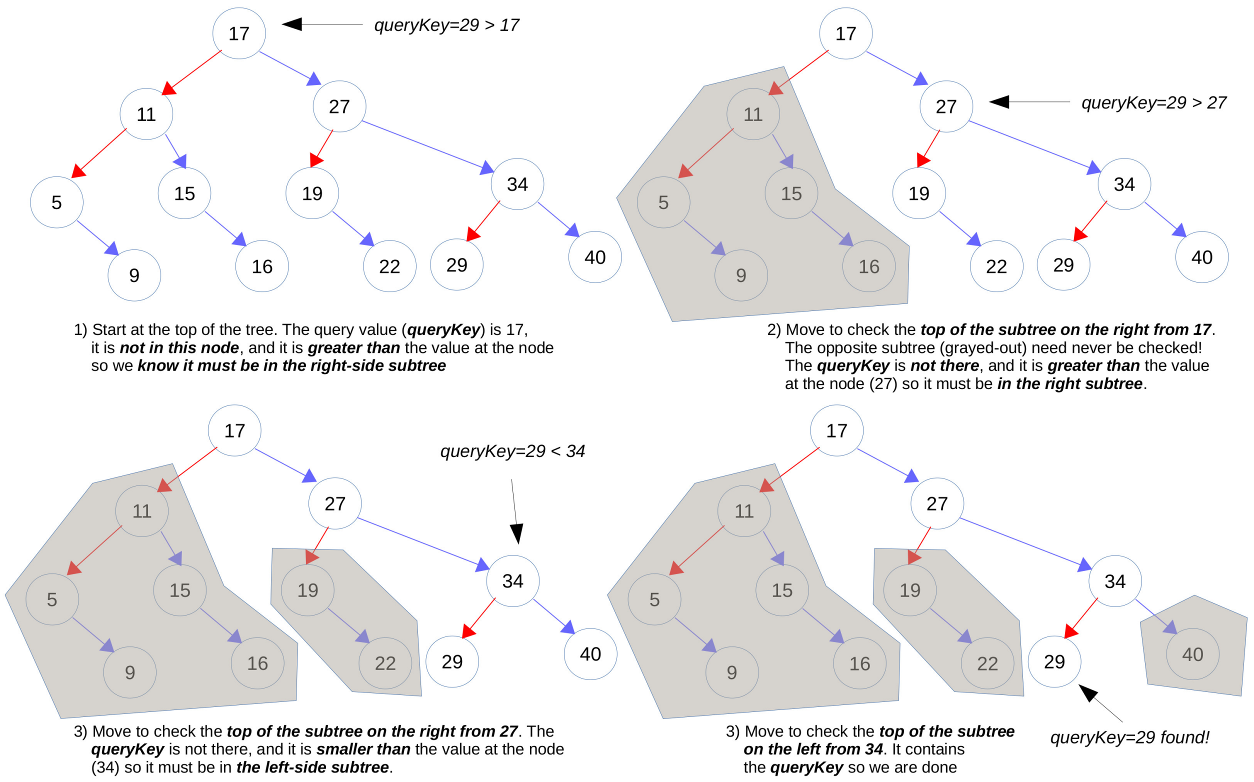
```
  // This function takes the root of a subtree <subtreeRoot>
  // and a key value we are searching for <queryKey>. If an
  // item in the tree contains a matching key, the function
  // returns the data contained in that node.

searchForKey <-- <subtreeRoot>, <queryKey>

if the <queryKey> is equal to the <key> at <subtreeRoot>
  // Found the key: Return the data stored at this node

  otherwise
      if the <queryKey> is less than the <key> in <subtreeRoot>
          // The node we want must be to the left of <subtreeRoot>
          searchForKey <-- <leftSubtreeRoot>, <queryKey>
      otherwise
          // The node we want must be to the right of <subtreeRoot>
          searchForKey <-- <rightSubtreeRoot>, <queryKey>
```

The search process starts at the **top of the tree**, checking the query value against the value stored at a node. If the value is found the search succeeds and returns the requested data item, otherwise, it **checks whether the query value should be in the left or the right subtree** (which is possible because of the **BST property**), and continues searching on the corresponding subtree.

The search process is illustrated in Fig. 4.11.

Note

With **BSTs**, we normally call the values used to search for information in the tree **search keys**. If the tree contains data items which are simple data types then the keys and the data values are the same. But remember that we intend to use these data structures to organize and maintain large collections of compound data types. So in general, **the key will be a suitable field**, or a **subset of fields** from the **CDTs**. **Keys have to be comparable** (i.e. besides checking for equality, **we must be able to tell which of the two keys is greater and which is lesser** according to some ordering that makes sense for our data). This often involves **writing a comparison function** that works with our data type. Finally, as we discussed in the previous Chapter, **search keys must uniquely identify each item in a collection**.



queryKey=29 > 17
1) Start at the top of the tree. The query value (***queryKey***) is 17, it is ***not in this node***, and it is ***greater than*** the value at the node so we ***know it must be in the right-side subtree***
queryKey=29 > 27
2) Move to check the ***top of the subtree on the right from 17***. The opposite subtree (grayed-out) need never be checked! The ***queryKey*** is ***not there***, and it is ***greater than*** the value at the node (27) so it must be ***in the right subtree***.
queryKey=29 < 34
3) Move to check the ***top of the subtree on the right from 27***. The ***queryKey*** is not there, and it is ***smaller than*** the value at the node (34) so it must be in ***the left-side subtree***.
3) Move to check the ***top of the subtree on the left from 34***. It contains the ***queryKey*** so we are done
queryKey=29 found!

Figure 4.11: Search in a **BST** involves checking nodes starting at the top, at each step deciding whether we have found the data we are looking for (**the queryKey**) or whether we should look for it in the **right subtree** or **the left subtree**. Nodes shaded in gray will never be checked because they belong in a subtree where we know the **queryKey** could never be placed.

You can see how, similarly to binary search, the search process in a **BST** quickly discards from search a large portion of all the values stored in the tree. The question is **how many items need to be examined for us to find the query key (or determine it's not in the tree)?**.

The search moves **down one level in the BST for every comparison** between the **queryKey** and values in the tree. This means that at most, the **search has to examine a number of nodes equal to the height of the BST** - the number of levels in the tree. The **height** of a **BST** is defined as **the length of the longest path from the root of the tree to a leaf node**. This is the number we need to know in order to estimate how muck work is needed to find a specific item in a **BST**.

So, **how many levels are there in a full BST?** To answer that question we need to figure the **maximum height** and **the minimum height** that a **BST** with **N** items in it could have.

The **height** of the tree depends on where data items are relative to each other. To understand how the tree is shaped, we need to take a look at how the tree is built by **inserting data items** in some order into an initially empty **BST**.

4.4.2 Inserting nodes into a BST

For **linked lists**, we had to keep around a pointer to the **head** of the list. In the case of a **BST**, we will need to keep around a pointer to the **root** of the tree (the node at the very top). The insert process **must ensure that the BST property is enforced** when a new item is inserted in the tree.

The pseudocode for the **BST insert** operation is shown below:

```
   // Given the <root> of the BST
   // and a new data item whose key is <newKey>
   // to be inserted in the tree.

   if the BST is empty (root is NULL)  // The first key inserted in the
      <root> := <newKey>               // tree becomes ROOT
   otherwise
      BST_insert <-- <root>, <newKey>

   // The insert function takes the root of some
   // subtree (or the whole tree) and the newKey
   // and inserts the new node in the right place
   // so the BST property is preserved

 BST_insert <-- subtreeRoot, newKey

      if the <newKey> is equal to the <key> at <subtreeRoot>
         then this is a duplicate key: Return without inserting anything

      otherwise

         if the <newKey> is less than the <key> in <subtreeRoot>
            // <newKey> should be on the left-side from <subtreeRoot>
            if the <leftSubtree> of <subtreeRoot> is empty
                  <leftSubtreeRoot> := <newKey>
            else
               BST_insert <-- <leftSubtreeRoot>, <newKey>

         otherwise
            // <newKey> should be on the right-side from <subtreeRoot>
            if the <rightSubtree> of <subtreeRoot> is empty
                  <rightSubtreeRoot> := <newKey>
            else
               BST_insert <-- <rightSubtreeRoot>, <newKey>
```

The **insert** operation works very similarly to the **search** operation. It starts at the top of the tree, working its way downward and choosing either the left or the right subtree at each level depending on whether the **new key** is **less than** or **greater than** the key at each node visited. Once it finds **an empty subtree**, it inserts the new node there. Fig. 4.12 shows several examples of this process.

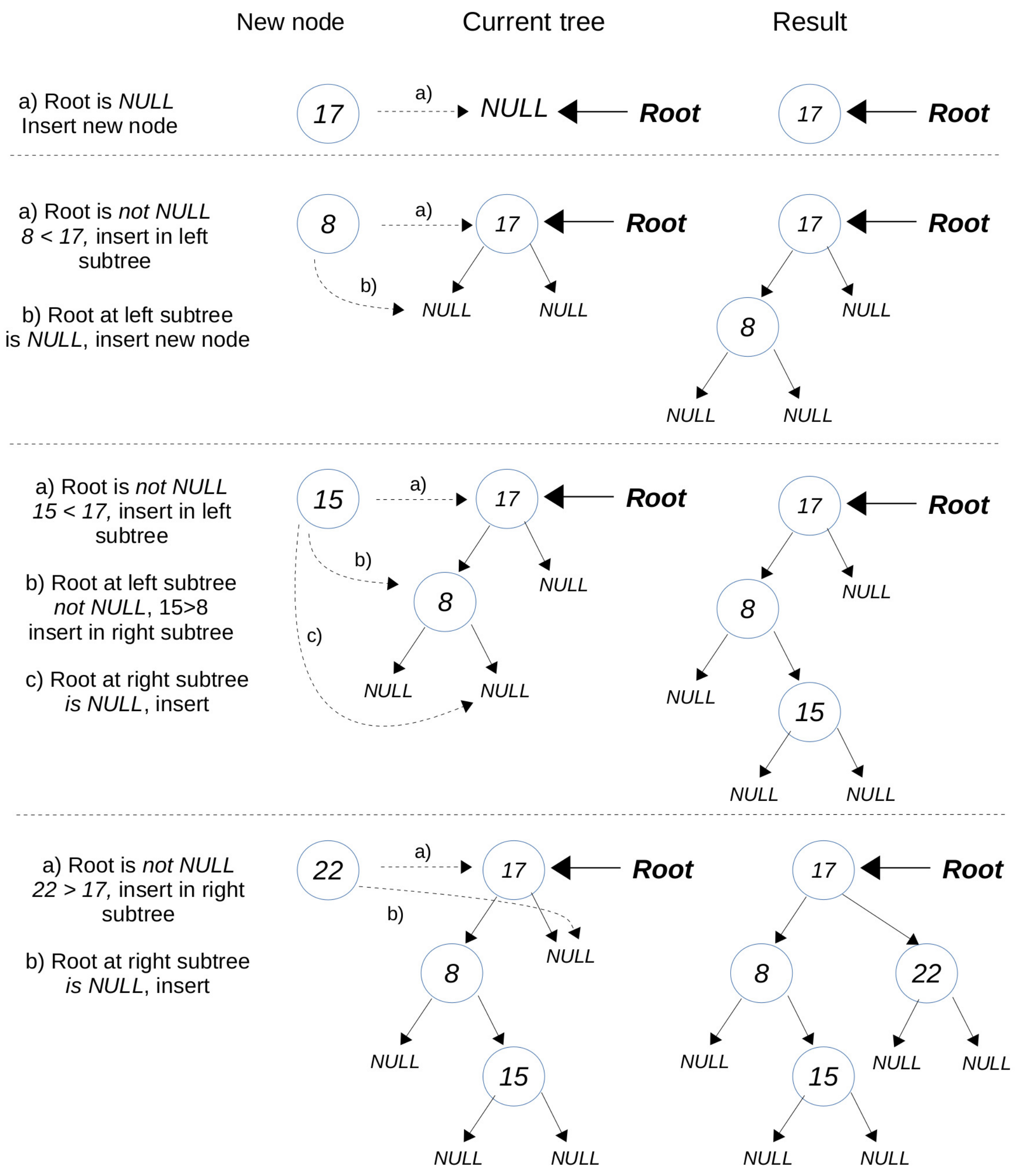

New node
Current tree
Result
a) Root is NULL
Insert new node
a) Root is not NULL
8 < 17, insert in left subtree
b) Root at left subtree is NULL, insert new node
a) Root is not NULL
15 < 17, insert in left subtree
b) Root at left subtree not NULL, 15>8 insert in right subtree
c) Root at right subtree is NULL, insert
a) Root is not NULL
22 > 17, insert in right subtree
b) Root at right subtree is NULL, insert
Root
NULL

Figure 4.12: Examples of **BST insert** starting with an empty tree. The process works its way down from the top of the tree, choosing the left or right subtree as is appropriate at each level, until an empty subtree is found where the new data item should be stored.

✍ **Exercise 4.4** Starting with the last BST in Fig. 4.12 (after 22 is inserted), list the steps carried out, and draw the resulting tree, after inserting: 19, 4, 1, 6 , and 21 (in that order)

With this in mind, let's revisit our question regarding the possible values for the **height** of a BST that contains **N** items. Because of how the insertion process works, the **shape** of the **BST** (the location of the nodes) will be dependent on **the order** in which keys are inserted. To see this, have a look at the example below, which inserts the numbers **1** to **7** in an **initially empty BST**.

Example 4.2

```
Let's first see what happens when we insert the numbers 1-7 in order into an intially
empty BST.

    1      1      1     1    .. etc .. 1
            \      \     \              \
             2      2     2              2
                     \     \              \
                      3     3              3
                             \              \
                              4              4
                                              .
                                               .
                                                \
                                                 7

Because the numbers are inserted in order, each key we add must be placed to the right
of the one that was inserted just before. The resulting BST is completely one-sided
and in fact looks like a linked list!

The height of a BST that results from inserting N items in order is N-1

Now let's see what happens when we insert the same numbers in the order
4, 2, 6, 1, 3, 5, 7

4      4      4       4       4        4         4
      /      / \     / \     / \     /   \     /   \
     2      2   6   2   6   2   6   2     6   2     6
                   /       / \     / \   /   / \   / \
                  1       1   3   1   3 5   1   3 5   7

In this case, the keys are inserted in such a way that the resulting BST is
balanced - which means that there are no large differences in height between
different subtrees at any place. This latter tree is an example of a
'complete' BST.
```

So the **height** of the **BST** depends on the order in which items are added to the tree. From the example above we can immediately tell that **in the worst case** the **height of the BST** will be **N-1** which means that both **search** and **insert** have a **worst case Big O complexity** of $O(N)$. That should not be surprising because in the worst case, the **BST** looks like a **linked list**.

Conversely, when the tree is as **short as possible**, the height of the tree is given by $\lfloor log_2(N) \rfloor$. This follows from the fact that each successive level in the tree contains **twice as many** nodes as the one before - or what amounts to the same thing: **each additional level in the tree doubles the number of nodes it can contain**. This can only happen when the **BST** is **complete**, which means that **every level of the tree is full with the possible exception of the last one**, and **any nodes in the last level are packed to the left, with no gaps in between**.

Therefore the **the Big O complexity** of **search** and **insert** is $O(log_2(N))$ on a **complete BST**.

At this point let us remember one **important assumption** regarding our collection: Items are added **in random order**. This means we would have to be very very lucky (or very very unlucky) to get either of these special cases of **BSTs**. Most of the time we have a tree whose height is between that of a **complete BST** and that of a misshapen tree that looks like a linked list.

In order to get a useful estimage of what we can expect from a typical **BST**, we need to know what is the **average height of a BST** that results from items being added in random order. The surprising, but very helpful result is that **the average height of a BST** built from a random sequence of insertions is $O(log_2(N))$ - it is not **exactly** $log_2(N)$ but it is within some constant factor of it. Detailed proofs for this result can be found in standard textbooks on algorithms.

This is a good result because it means that **on average**, the **Big O** complexity of **search** and **insert** on a **BST** is $O(log(N))$. And now we're getting somewhere! in the average case, **BST search** is as efficient as **binary search**, and **we do not have to sort the data** because the **BST insert** operation (which is also $O(log(N))$) ensures the **BST property holds** everywhere in the tree at all times.

Note

We can not trick the Universe. The total computational cost (on average) of building a **BST with N keys** is **O(**$N \cdot log(N)$**)** because we are performing **N** insert operations, each with a cost of $O(log(N))$. This is just the same as the cost of sorting an array so we can use binary search. **However**, once we have the **BST** we can **insert** new items into it at a cost of $O(log(N))$, whereas to insert a new item into the sorted array we would need to use **insertion sort** which has a complexity of $O(N)$ or re-sort the array at a cost of $O(N \cdot log(N))$. The real advantage of the **BST** is not in the **initial setup** but in the fact it allows us to perform **any succeeding operations efficiently**.

✍ **Exercise 4.5**

Draw the **BST** that results from inserting the following keys in sequence: 49, 85, 22, 15, 97, 67, 4, 71, 35, 8, 99, 64, 2, 17, 69, 50

4.4.3 Deleting items from a BST

The **delete** operation is a bit more interesting than both **search** and **insert**. This is because we must ensure that **after deletion the BST property still holds** everywhere in the tree.

There are three cases we must consider for deletion:

Case a) The item to be deleted is **a leaf node** (it has **no children**). Figure 4.13 shows an example of this case. The process to be carried out is:

- Find the node with the item we want to **remove** - same process as **BST search**.
- Since this a **leaf node**, there is **nothing else** in its **subtree**. Once we remove this node, there is nothing left.
- Therefore, we set the corresponding **pointer in the parent node to NULL** to indicate that there is now an **empty subtree** on the side where the deleted item used to be.

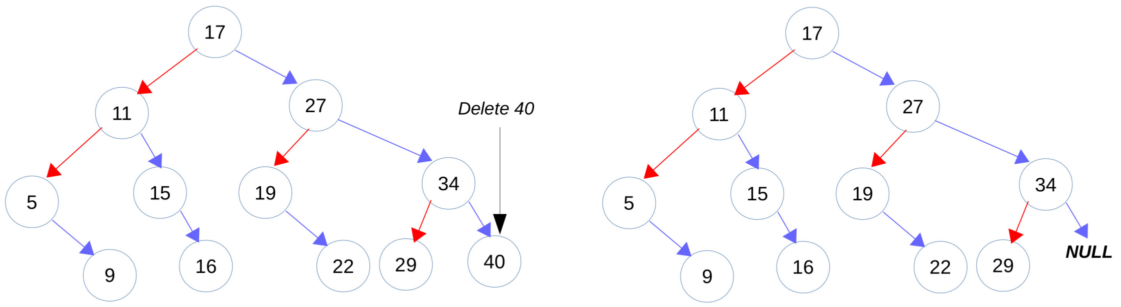


Figure 4.13: Example of **case a)** of **BST delete**. The item being deleted (40) is in a **leaf node**, therefore we simply delete the node and set the corresponding pointer **in the parent node** to **NULL**.

Case b) The item to be deleted has **one child only**. This is very similar to deleting an item inside a **linked list** as shown in Fig. 4.14. The process to be carried out is:

- Find the node with the item we want to **remove** - same process as **BST search**.
- The node's child is **guaranteed to be on the correct side with regard to the node's parent** because **BST insert** ensures the **BST property holds** throughout the whole tree.
- Therefore, we have to link **the node's parent** to **the node's child** and then delete the node.

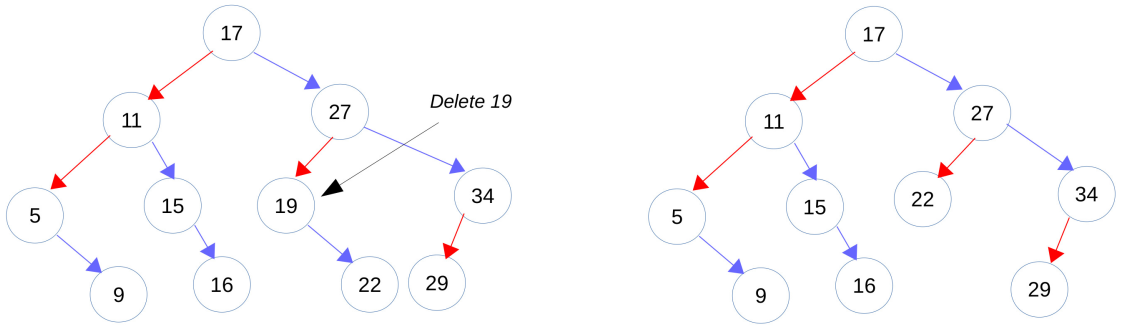


Figure 4.14: Example of **case b)** of **BST delete**. The item being deleted (19) has a single child (22). Linking the **parent node (27)** to the **child node (22)** preserves the **BST property** and we can then delete (19).

Case c) The most general case, the item to be deleted has **two children**. In this case, deleting the node and trying to re-link the tree results in a problem: we would have two dangling nodes (the children of the node we just deleted) and only one link where to attach them. Rather than try to figure out where to attach the two children while preserving the **BST property**, we can solve this problem by carefully **moving data items inside the tree** as shown in Fig. 4.15. The process is as follows:

- Find the node with the item we want to **remove** - same process as **BST search**.
- Search in **the right-side subtree** for the **successor** of this node - this is **the smallest key** in the **right subtree** of the node we are deleting. The **successor** is easily found by starting at the top of the **right subtree** and then traversing **as far down and to the left** as possible (i.e. without following any right-child links).
- **Promote** the **successor** to the **node we are deleting** - this means copying the data (and key) stored at the successor node into the node we are deleting.
- Finally, **delete the successor node** from **the right subtree** - this will be either **case a)** or **case b)**.

The point of this process is that we **do not remove internal nodes** in the **BST**. Instead, we move data around such that we are left with **a valid BST** (the BST property holds everywhere after we have moved the data), and **an easier case of deletion** either **case a)** or **case b)** since the **successor node** can **never have two children** (think through the process of finding the successor to see why that has to be the case).

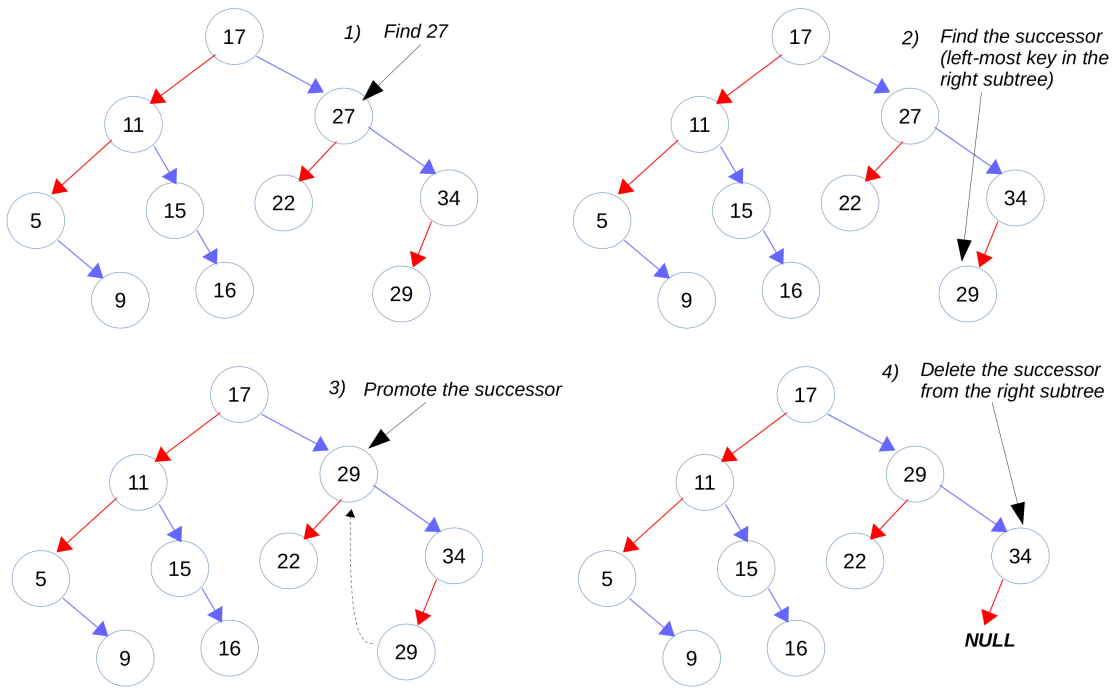

1) Find 27
2) Find the successor (left-most key in the right subtree)
3) Promote the successor
4) Delete the successor from the right subtree
NULL

Figure 4.15: Example of **case c)** of deletion. The node has two children so we do not want to remove it, instead we **promote** the **successor** (smallest key in the right subtree) to the node we want to delete, and then **delete the successor** from the **right subtree**.

Note

It should be noted that deletion would work equally well if we used the **predecessor** (largest key in the left subtree) instead of the **successor**. Both of these options will result in a valid **BST** (the **BST property** is maintained throughout the tree), and they involve the same amount of work in terms of computational complexity - there is **no difference**. For **consistency**, in this book we will use the **successor** when deleting nodes from a **BST**.

✍ **Exercise 4.6** Beginning with the tree in Fig. 4.15 after (29) was deleted, draw the trees resulting from deleting: 9, 29, 22, and 11 in that order. Be sure to identify which case of deletion is needed for each number.

Question: What is the **average case Big O complexity** of deleting a node from a **BST**? The process of **finding the node we want to delete** is $O(log(N))$, it is just a regular **BST search**. The actual deletion requires a **fixed number of operations** for cases **a) and b)**. For **case c)** it involves **one additional search with cost** $O(log(N))$ to find the node's **successor**. So just like **search** and **insert**, the **average case Big O complexity of delete** is $O(log(N))$.

This last is important. It means that we can expect all **BST** operations to be **as efficient as binary search** in the average case.

Note

In practice, **BSTs** perform incredibly well under general conditions, but you may be concerned that they perform well only in the **average case**. It's worth pointing out that there are several other types of trees such as **AVL-trees** and **B-trees** that have the property that they **remain balanced** regardless of the order in which keys are inserted. Such **balanced trees guarantee** that the **Big O complexity** of **search, insert**, and **delete** operations remains $O(log(N))$ in the **worst case**. You can learn about these more sophisticated trees in any standard algorithms book.

4.5 Tree traversals

In linked lists, an implicit operation that needs to be carried out frequently is a **list traversal**. A full list traversal involves visiting **each node in the list exactly once**. There are many situations in which we need to visit nodes in a binary tree (for example, in a tree that stores numbers we may want to compute the average of the values in the tree). Because in **BSTs** there is more than one link at each node, that means there is a choice in terms of **the order in which we visit** the nodes to carry out whatever work we need to do.

The process of visiting each node in a **BST** exactly once is known as a **tree traversal**, and because there is a choice in terms of the order in which we visit nodes and work with the data stored in them, there are **3 different types** of **tree traversals** for **binary trees**.

4.5.1 In-order traversal

Possibly the most common type, it specifies nodes will be visited in the order given by the following procedure, starting at the **root** of the **BST**

At any particular node i:

- 1) Perform an **in-order traversal** of the left subtree
- 2) Carry out the specified work at node i
- 3) Perform an **in-order traversal** of the right subtree

The process is illustrated in Fig. 4.16. For this figure, the **work being done** at each node consists very simply of **printing the value in the node**. But in general this could involve more complicated processing of whatever data is stored in the corresponding **BST** node. What is important in the figure is understanding the **how the steps of the process** are applied at each node, and the **order in which nodes are processed**.

In-order traversal visits nodes **in sorted order** with respect to their **key value**. So in the sample case above, the process will print a **sorted list of keys** in the tree: 5, 11, 15, 17, 19, 27, 34.

What is the Big O complexity of in-order traversal? Just like **list traversal**, a **tree traversal** will need to visit each node in the tree. If there are **N** nodes, that means the traversal process will be $O(N)$.

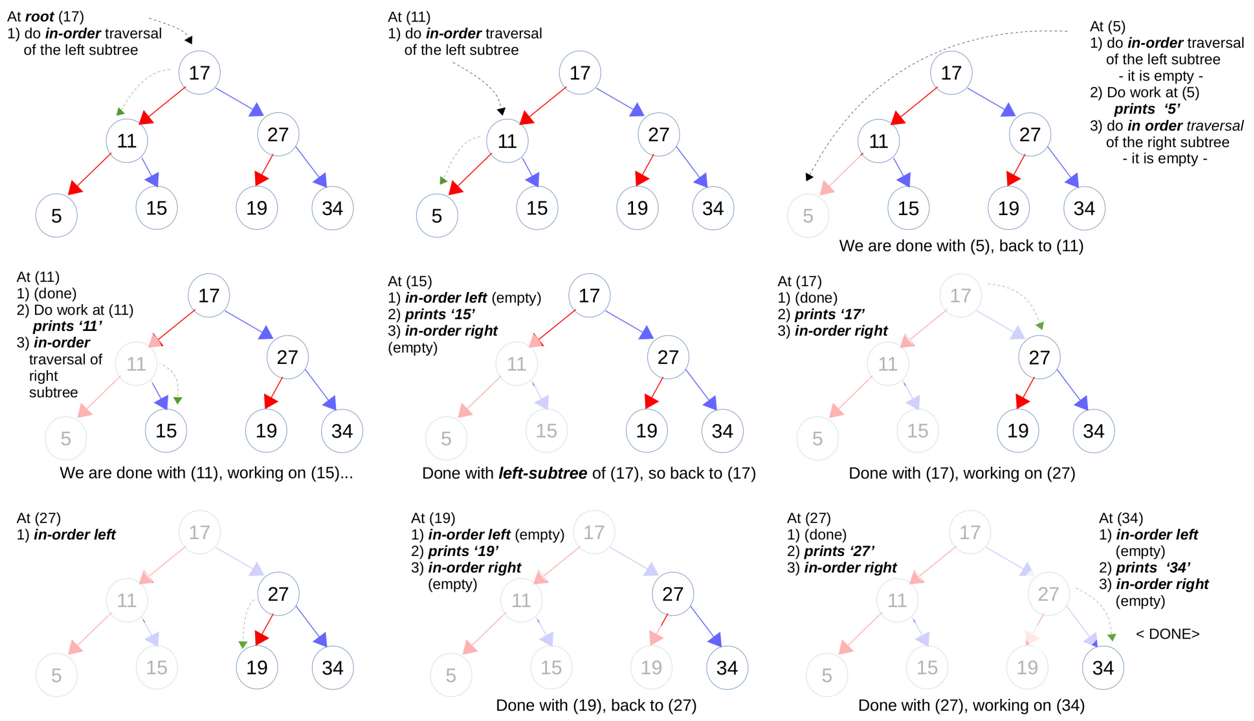

At root (17)
1) do in-order traversal of the left subtree
At (11)
1) do in-order traversal of the left subtree
At (5)
1) do in-order traversal of the left subtree - it is empty -
2) Do work at (5) prints '5'
3) do in order traversal of the right subtree - it is empty -
We are done with (5), back to (11)
At (11)
1) (done)
2) Do work at (11) prints '11'
3) in-order traversal of right subtree
We are done with (11), working on (15)...
At (15)
1) in-order left (empty)
2) prints '15'
3) in-order right (empty)
Done with left-subtree of (17), so back to (17)
At (17)
1) (done)
2) prints '17'
3) in-order right
Done with (17), working on (27)
At (27)
1) in-order left
At (19)
1) in-order left (empty)
2) prints '19'
3) in-order right (empty)
Done with (19), back to (27)
At (27)
1) (done)
2) prints '27'
3) in-order right
Done with (27), working on (34)
At (34)
1) in-order left (empty)
2) prints '34'
3) in-order right (empty)
< DONE>

Figure 4.16: Step-by-step **in-order** traversal in a sample **BST**, note the order in which the three steps of the process are applied at each node. In particular, nodes whose **left subtree** is **not empty** have to wait until their left subtree has been processed before they get their turn.

Note

In-order traversal of a **BST** can be used for sorting. The process is called **tree sort**.

- Insert all items to be sorted into the **BST** using as **key** whatever data field or fields we want the data sorted by
- Perform an **in-order** traversal of the resulting tree - this produces the **sorted list of items**

What is the worst-case Big O complexity of tree sort?. We can figure this out by accounting for the **Big O** complexity of each of the two steps in the tree sort process.

- Inserting items into the **BST**: In the **worst case** the items are already sorted, and inserting them into the **BST** produces a structure that resembles a **linked list**. The cost of **each insert** is $O(N)$ because each new item will go **to the bottom of the growing tree**, and we perform N insertions which gives a complexity of $O(N^2)$ for building the **BST**
- The **in-order** traversal has a cost of $O(N)$ regardless of the shape of the tree

This gives a total cost of $O(N^2 + N)$ and as we saw above, the fastest-growing term will dominate this complexity bound, so the **worst case Big O complexity of tree sort is** $O(N^2)$. Which is the same as **bubble sort** and **not very good**.
However, if the input is **not sorted**, as we might expect if data arrives in **random order**, things look a lot better:

- Inserting items into the **BST** will in the **average case** have a computational cost of $O(N \cdot log(N))$
- The **in-order** traversal has a cost of $O(N)$

This gives a total cost of $O(N \cdot log(N) + N)$ and as usual the fastest-growing term dominates the complexity bound. So the **average case** cost of **tree sort** is $O(N \cdot log(N))$ which is on par (in terms of **Big O complexity**) with the best known sorting methods. If instead of a **BST** we use a **balanced tree such as an AVL**, then the $O(N \cdot log(N))$ cost is **guaranteed in the worst case**.

4.5.2 Pre-order traversal

One of the most common uses of a **pre-order traversal** is making an exact duplicate of a **BST**. By traversing the **BST** in **pre-order**, making a copy of each data item, and inserting the new item into another (initially empty) **BST** we obtain two identical trees - each corresponding node is in exactly the same location with respect to the root of the tree. Other applications include **search algorithms** on graphs, which we will study in the next chapter. The **pre-order** traversal is as follows:

At any particular node i:
- 1) Carry out the specified work at node i
- 2) Perform a **pre-order traversal** of the left subtree
- 3) Perform a **pre-order traversal** of the right subtree

The process is illustrated in Fig. 4.17. Same as for the **in-order** example, the work being done at each node is simply printing the value at the node. In this example, the traversal will print: 17, 11, 5, 15, 27, 19, 34. Which is pretty different from what we got out of the **pre-order** traversal.

4.5.3 Post-order traversal

The last type of traversal for binary trees is the **post-order** traversal. It has applications in **compilers** and **parsing languages**. It is also essential for managing **BSTs** because whenever we want to **delete a BST** this has to be done in **post-order**. **Post-order** traversal consists of:

At any particular node i:
- 1) Perform a **post-order traversal** of the left subtree
- 2) Perform a **post-order traversal** of the right subtree
- 3) Carry out the specified work at node i

The process is illustrated in Fig. 4.18. As in the previous traversal examples, the work being done at each node is simply printing the value at the node. **Post-order** traversal will print: 5, 15, 11, 19, 34, 27, 17. Note that **the parent node has to wait** until **both of its subtrees have been processed**, which is what we want when **deleting a BST** - we can not delete a node until both of its subtrees have been removed.

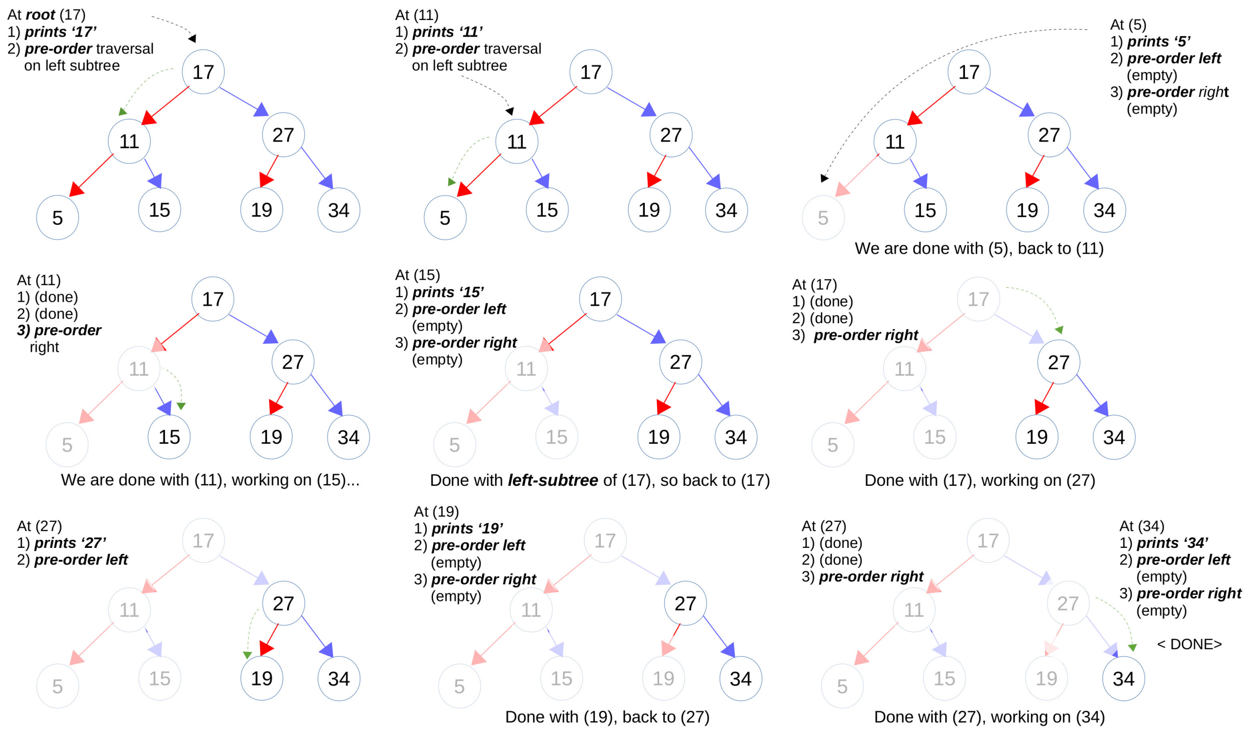

At root (17)
1) prints '17'
2) pre-order traversal on left subtree
At (11)
1) prints '11'
2) pre-order traversal on left subtree
At (5)
1) prints '5'
2) pre-order left (empty)
3) pre-order right (empty)
We are done with (5), back to (11)
At (11)
1) (done)
2) (done)
3) pre-order right
We are done with (11), working on (15)...
At (15)
1) prints '15'
2) pre-order left (empty)
3) pre-order right (empty)
Done with left-subtree of (17), so back to (17)
At (17)
1) (done)
2) (done)
3) pre-order right
Done with (17), working on (27)
At (27)
1) prints '27'
2) pre-order left
At (19)
1) prints '19'
2) pre-order left (empty)
3) pre-order right (empty)
Done with (19), back to (27)
At (27)
1) (done)
2) (done)
3) pre-order right
At (34)
1) prints '34'
2) pre-order left (empty)
3) pre-order right (empty)
< DONE>
Done with (27), working on (34)

Figure 4.17: Step-by-step **pre-order** traversal in a sample **BST**. In this traversal method, the work at the node is done **before** either of the **subtrees** is processed.

4.6 Implementing a BST

Now that we have learned about all the operations that a **BST** supports, as well as the different **tree traversal** methods we can perform on a binary tree, the last step is to turn our **BST ADT** into a **data structure** by implementing all of the **BST operations** in **C**. The implementation of each function is listed below, you should read through it carefully and review the **pseudocode** and **examples** of the different operations as you work your way through the program code.

Firstly, let's define the **CDTs** we will need. We are going to work with the same example application as in the previous chapter (a collection of restaurant reviews), so we need a **CDT** for the **reviews**, and another one for the **BST nodes**.

```
typedef struct Restaurant_Score
{
  char restaurant_name[MAX_STRING_LENGTH];
  char restaurant_address[MAX_STRING_LENGTH];
  int score;
}Review;

typedef struct BST_Node_Struct
{
  Review rev;      // Stores one review
  struct BST_Node_Struct *left; // A pointer to its left child
  struct BST_Node_Struct *right; // and a pointer to its right child
} BST_Node;
```

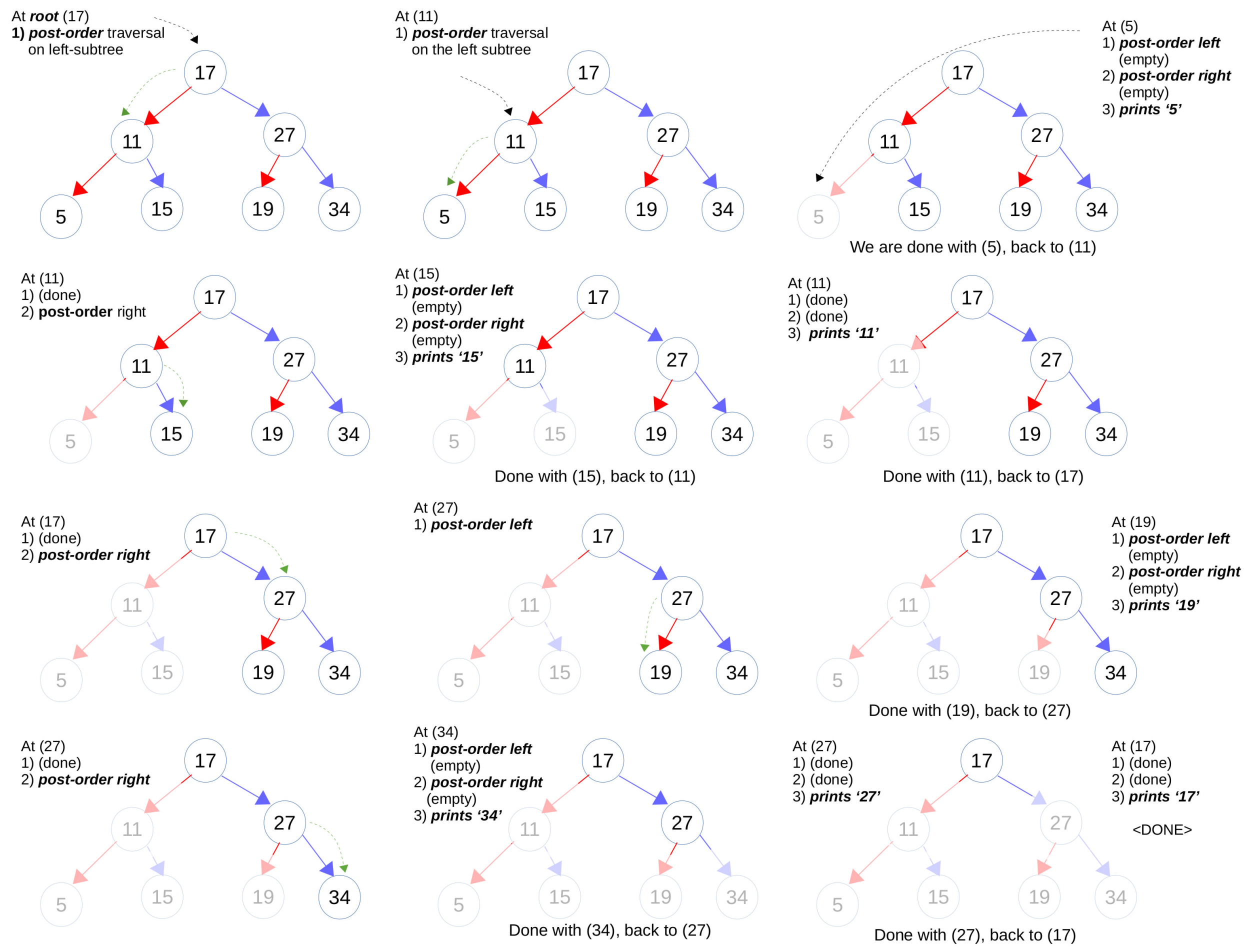

At root (17)
1) post-order traversal on left-subtree
At (11)
1) post-order traversal on the left subtree
At (5)
1) post-order left (empty)
2) post-order right (empty)
3) prints '5'
We are done with (5), back to (11)
At (11)
1) (done)
2) post-order right
At (15)
1) post-order left (empty)
2) post-order right (empty)
3) prints '15'
Done with (15), back to (11)
At (11)
1) (done)
2) (done)
3) prints '11'
Done with (11), back to (17)
At (17)
1) (done)
2) post-order right
At (27)
1) post-order left
At (19)
1) post-order left (empty)
2) post-order right (empty)
3) prints '19'
Done with (19), back to (27)
At (27)
1) (done)
2) post-order right
At (34)
1) post-order left (empty)
2) post-order right (empty)
3) prints '34'
Done with (34), back to (27)
At (27)
1) (done)
2) (done)
3) prints '27'
Done with (27), back to (17)
At (17)
1) (done)
2) (done)
3) prints '17'
<DONE>

Figure 4.18: Step-by-step **post-order** traversal in a sample **BST**. In this traversal method, **both subtrees of a node** are processed before the node itself.

The **Review CDT** should be perfectly familiar to you by now. The **BST_Node CDT** is just a small modification of the **Review_Node CDT** we used for linked lists. The only difference is that instead of **a single** *next* **pointer**, we now have **two pointers** to **BST_Node CDTs**. One for the **left child**, and one for the **right child**. By convention, in most **BST** implementations you will ever come across, these pointers are called **left** and **right**.

Next we have to write a function to **allocate and return** a newly created **BST_node**.

```
BST_Node *new_BST_Node(void)
{
  BST_Node *new_node=NULL;  // Pointer to the new node

  new_node=(BST_Node *)calloc(1, sizeof(BST_Node));
    if (new_node==NULL)
    {
        printf("new_BST_Node(): Error, there is not enough memory to create new node, returning
            NULL\n");
        return NULL;
    }

  // Initialize the new node's content (same as with linked list)
  new_node->rev.score=-1;
  strcpy(new_node->rev.restaurant_name,"");
  strcpy(new_node->rev.restaurant_address,"");
  new_node->left=NULL;
  new_node->right=NULL;

  return new_node;
}
```

This function is very similar to the one we wrote for **linked lists**, nothing new here except for having to initialize two pointers rather than one.

Next up is the function that inserts a new item into the **BST**:

```
BST_Node *BST_insert(BST_Node *root, BST_Node *new_node)
{
    // This function takes as input the <root> of a subtree
    // (it can be any subtree, or the whole BST), and a
    // <new_node> that contains a review (already filled
    // with valid information).
    // It inserts the <new_node> into the BST

  if (root==NULL)   // Tree is empty, new node becomes
    return new_node; // the root

  // Check for duplicates (and refuse to insert duplicates)
  if (strcmp(new_node->rev.restaurant_name,\
      root->rev.restaurant_name)==0)
    {
        printf("BST_insert(): Duplicate data item detected. Ignoring\n");
        return root;      // The root did not change
    }

  // This node is occupied, we need to figure out on which
  // side of this node the <new_node> needs to go, and then
  // go and insert the <new_node> in that subtree.
```

```
if (strcmp(new_node->rev.restaurant_name,\
    root->rev.restaurant_name)<=0)
{
      // <new_node>'s key is less than the key at <root>,
      // it must be inserted in the left subtree

      root->left=BST_insert(root->left,new_node);

      // The situation at this point is as shown below:
      //
      //              /
      //             /
      //           root       <-- we're working here
      //           /
      //          /
      //     leftRoot        <-- The <new_node> should
      //                          go in this subtree
      //
      // leftRoot is stored in 'root->left'
      //
      // So, we tell the BST_insert() function to go and
      // insert the <new_node> in the subtree that
      // starts at 'root->left'.
      //
      // Why are we updating 'root->left' with the return
      // value of BST_insert()?
      //
      // Because if the situation is as follows:
      //
      //          /
      //         /
      //       root
      //       /
      //      /
      //    NULL
      //
      // Then the first check in BST_insert will simply
      // return <new_node> because this node will
      // become the top of a new subtree. Updating
      // 'root->left' results in
      //
      //         /
      //        /
      //      root
      //       /
      //      /
      //   new_node
      //
      //  If 'root->left' is NOT NULL when we call
      //  BST_insert(), nothing will happen as
      //  BST_insert() will simply return the same
      //  'root->left' it received.
}
else
{
      // <new_node>'s key is greater than the key at <root>
      // it must be inserted in the right subtree

  root->right=BST_insert(root->right,new_review);
```

```
    // Which works exactly as shown above, but on the
    // subtree that is on the right side of <root>
  }

  return root;  // Return the same <root> we received
}
```

The only thing worth commenting in the code above is that the **BST_insert()** function is that we use it **within the function itself** to get the work done. This is an example of **recursion**. We will take a detailed look at **recursion** as a general tool for solving a very large variety of problems that have particular characteristics. In this case, we observe that **inserting a node into a tree** requires us to be able to **insert a node into a subtree** (the left or right one, depending on the key). But every **subtree of a BST** is also **a BST** (see Fig. 4.19).

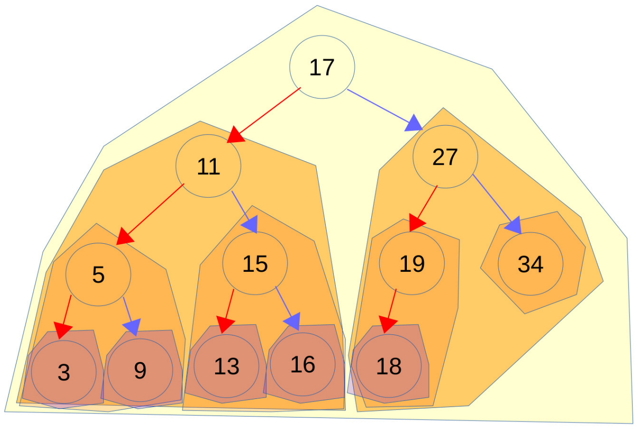


Figure 4.19: Each **subtree** in a **BST** is itself a fully valid **BST** - so inserting a key in a particular **BST subtree** is exactly the same thing as inserting a key in a **BST**.

We take advantage of this property to use the same **BST_insert()** function to handle the process as we traverse from the **root of the BST** to the place where the new node should go.

Because **C** reserves space for a function's variables at the moment we call the function (see Chapter 2), each **call to BST_insert()** will have **its own reserved space for input arguments, local variables, and return value**. The structure of the function shown above directly matches the pseudo-code description of how **BST** insert is supposed to work, there really is nothing special about **recursion**, it is the natural way to implement operations on trees.

The next function to implement is **search**:

```
BST_Node *BST_search(BST_Node *root, char name[1024])
{
    // The function takes as input the <root> of some
    // subtree in the BST (can be the root of the whole
    // tree).
    // It searches for the node that contains the matching
    // restaurant name, and if found, it returns a pointer
    // to the node containing the matching review.

  if (root==NULL) return NULL; // We reached an empty
                                // subtree, the queryKey
                                // is not in the BST
  // Check if this node contains the review we want, if so, return
  // the pointer to this node
  if (strcmp(root->rev.restaurant_name, name)==0)
    return root;

  // The queryKey is not in the current node, decide
  // if we need to search in the left subtree or
  // in the righ subtree.

  if (strcmp(name, root->rev.restaurant_name)<=0)
  {
        // queryKey is less than the key at <root>
        // so we must go search in the left subtree

        return BST_search(root->left,name);

        // The call above performs a search on
        // the left subtree, and will return
        // whatever we find there i.e. NULL if
        // the queryKey is not found in the
        // left subtree, or the pointer to the
        // node that contains the matching
        // review, whereved it may be in the
        // left subtree.
  }
  else
  {
        // queryKey is greater than the key at <root>
        // so we must go search in the right subtree

    return BST_search(root->right,name);
  }
}
```

Not surprisingly, **BST_search()** is also **recursive**. But if you work your way through it, you will see it is simply a direct implementation of the pseudo-code for **search** that we studied above.

The last operation we need to implement for a fully working **BST** is the **insert** operation:

```
BST_Node *BST_delete(BST_Node *root, char name[1024])
{
  // Takes as input the <root> of a subtree, and
  // the key of a node we want to remove. In this case
  // a restaurant's name.

  BST_Node *tmp;

  if (root==NULL) return NULL; // Tree or sub-tree is empty
                                // nothing to do.

  // Check if this node contains the review we want to delete

  if (strcmp(name,root->rev.restaurant_name)==0)
  {
        // Determine which of the three cases of
        // deletion we are dealing with.

    if (root->left==NULL && root->right==NULL)
  {
    // Case a), no children. The parent will
    // be updated to have NULL instead of this
    // node's address, and we delete this node

    free(root);
    return NULL;

    // The situation here is as follows:
    //
    //           parent
    //          /
    //         /
    //       root      <-- We are at this node
    //
    // We need to delete the current node,
    // and because there are no children, the
    // 'parent' node will have an empty
    // subtree. So, the function returns
    // NULL so the parent can update its
    // subtree to look like so:
    //
    //          parent
    //         /
    //        /
    //      NULL
    //
  }
  else if (root->right==NULL)
  {
    // Case b), only one child, left subtree
    // The parent has to be linked to the left
    // child of this node, and we free this node

    tmp=root->left;    // Store the pointer to the
    free(root);        // child!
    return tmp;

    // The situation is as follows:
    //
    //           parent
    //           /
    //          /
    //        root   <-- We are at this node
    //        /
    //       /
    //     child
    //
    // We need to connect 'parent' to 'child'
    // and remove <root>. So, we store
    // 'child' in a temporary pointer
    // (once we delete <root> that pointer
    // would be gone!)
    //
    //          parent
    //          /
    //         /
    //      (deleted)
    //
    //    tmp-->child
    //
    // The function then returns 'tmp' which
    // allows the 'parent' node to update its
    // own link so point to 'child'
    //
    //          parent
    //          /
    //         /
    //      child
  }
  else if (root->left==NULL)
  {
    // Case b), only one child, right subtree
    // The parent has to be linked to the right
    // child of this node, and we free this node

    tmp=root->right;
    free(root);
    return tmp;

    // Same as the case above, but the child node
    // is to the right of <root>
  }
  else
  {
    // Case c), two children.
    // We need to find the successor to this
    // node, promote it (copy the data to this
    // node), and then delete the successor
    // from the right subtree.

    tmp=root->right;   // Top of the right subtree
    while (tmp->left!=NULL)   // Successor is the
             tmp=tmp->left;         // leftmost key

          // Promote the successor (copy the data over
          // to <root>)
          root->rev = tmp->rev;

          // And finally delete the successor in the
          // right subtree
          root->right=BST_delete(root->right,tmp->rev.name);

      return root;
  }
}

// The review we want to delete is not in this node,
// determine if it should be on the left or right
// subtree, and call delete on the corresponding
// subtree.

if (strcmp(name, root->rev.restaurant_name)<=0)
{
      // The review we want should be on the left
      // subtree

  root->left=BST_delete(root->left,name);
}
else
```

```
  {
        // The review we want should be on the right
        // subtree

    root->right=BST_delete(root->right,name);
  }
  return root;      // Nothing changed here,
                      // return the <root> we
                      // received.
}
```

Just like **insert**, and **search**, the **BST_delete()** is recursive. It follows the pseudo-code for the **insert** operation as discussed above. While it does a bit more work than either of the other operations, the fundamental process is identical: Starting at the top of the tree, traverse downward choosing either the left or the right subtree until we find the place where work needs to be done.

This completes the implementation of all the **BST operations** required to build, maintain, and use a **BST** in **C**.

✍ **Exercise 4.7** Modify the app you created in the previous Chapter to manage the restaurant reviews so that it uses **BSTs** instead of **linked lists**. The goal here is to see how you can achieve **the same functionality** with **different data structures**.

There are two ways to go about this:

- a) Modify the listing from the previous Chapter to use **BST Nodes** and BST functions using the code above
- b) Expand on the code above by adding a **main()** function providing the same functionality as the program from the previous chapter. You will also need to implement any extra functions (i.e. beyond insert, search, and delete) that were present in the linked-list version

As you are building the application: Follow the **test-as-we-develop** process described at the end of Chapter 3. Make sure each function you add to your program is thoroughly tested before moving on to the next thing.

After your implementation is complete: Add a comprehensive set of **full program tests** to your test driver as discussed in Chapter 3. After all your testing is complete, you should be able to confidently state the program you implemented is solid, works on reasonable input, and handles tricky or erroneous input in a reasonable fashion.

✍ **Exercise 4.8** Implement **the three types of tree traversals**. The traversal functions should print the restaurant name, and the review score for each visited node, in the order specified by the traversal process. Add options in **main()** so that the user can request each of the three types of traversal.

Follow the test-as-we-develop process described at the end of Chapter 3.

✍ **Exercise 4.9** **Crunchy!** As we know, if we are unlucky with the order in which items are inserted into a **BST**, we could end up with a **very unbalanced tree** in which some branches are much longer than others (and in the worst case, there is a single branch with all the keys in it).

Write **pseudo-code** for a process that can be used to take as input **an unbalanced BST**, and that will then create a new, **balanced BST** (and delete the old unbalanced one). Make sure that your design **follows the guidelines set at the end of Chapter 2**.

Once you are happy with your design, expand the program that handles restaurant reviews to provide an option to **re-balance the BST** and implement your process into a function that produces a **balanced BST** from whatever the **current BST** looks like.

Test-as-you-develop - write the function that balances a **BST**, and test it thoroughly before integrating it with the rest of your application. Then perform **full program testing**. Follow the testing guidelines at the end of Chapter 3. Your tests should include **verification that the resulting tree is balanced**.

How do we verify that a tree is balanced? there are several ways in which we can define **balanced**. But for simplicity, for this exercise we will define a **balanced BST** as one in which the **difference in height** between the **leaf node that is closest to root** and the **leaf node that is farthest from root** is less than **2**.

How do we find the minimum (or maximum) height of a leaf node? - that's for you to work out, but perhaps tree traversals could be handy...

4.7 Have we solved the problem?

At this point we have a good understanding of what **BSTs** can do, and how to implement them, but we have yet to determine whether all this work has helped us solve the problem of **finding a more efficient way to store, organize, and access a large collection of information**. We may suspect that the **BST** is likely to give us faster search because **its average search complexity is better** than the linked list's average search complexity, but we still have to tie a few loose ends:

a) The time it takes to build the BST compared to the time needed to build a linked list

- For **N** items, the linked list can be built in $O(N)$ time - each item is added at the head so no traversal is needed. That's pretty good!
- For the same **N** items, the **BST** can be built in $O(N \cdot log(N))$ time on average since insertions always happen at the bottom of the tree, and the tree has height $O(log(N))$

Advantage: Linked list

b) The time it takes to build a BST compared with the time needed to sort an array (so we can use binary search)

- Using **merge sort** an array can be sorted in $O(N \cdot log(N))$ time on the worst case
- As we saw above, the **BST** can be built in $O(N \cdot log(N))$ time on average

Advantage: Array - **merge sort guarantees** $O(N \cdot log(N))$ even in the worst case, while for **BSTs** the worst case is $O(N^2)$ - the worst case happens with data that is initially sorted or close to sorted, so it can come up on real-world situations.

c) Search complexity

- Sorted array: $O(log(N))$ worst case using binary search
- Linked list: $O(N)$ both average and worst case
- BST: $O(log(N))$ average case, $O(N)$ worst case

Advantage: The **BST and sorted array** on the average, **sorted array** in worst case. The linked list is **not competitive**, it is simply too slow on average.

d) Storage use

- Sorted array: Fixed size, **not suitable for growing/shrinking** collections as that would require re-allocating the whole array
- Linked list: Space usage is $O(N)$ - one node per item
- BST: Space usage is $O(N)$ - one node per item

Advantage: The **BST and linked-list**, because they only use space for items actually present in the collection, whereas the array has to **pre-reserve** space and we have the problem of estimating in advance **how much we may need** which is not trivial. A lot of the space reserved may go unused.

4.7.1 So what do we choose?

The point of going through all of this work is to help you see that there is a **fair number of factors you have to consider when choosing how to store and organize a collection of data**. Up to this point we have studied 3 different ways of storing items: **arrays**, **linked-lists**, and **BSTs**, and as shown above, each one of them has advantages and disadvantages. So how are we to choose?

a) Consider how large your collection is going to be:

- For very large collections that change frequently, you want to **choose the data structure** that gives you the **best Big O complexity** for common operations like insert, delete, and search.
- Consider the **space requirements**: For **items with small memory requirements** (e.g. just numeric data, like ints or floats, or small compound data types) it's **not unreasonable to pre-allocate a large array** even though entries in it may go un-used. Conversely, for items with a large memory footprint (e.g. compound data types with lots of data, or multi-media content like images, sound clips, etc.) we prefer a dynamic data structure.
- For **smaller collections**, you may want to choose based on **ease of implementation**. There is a trade-off between ease of implementation (arrays require less complicated code to work with than **BSTs**) since it saves developer time, and is likely to produce code with smaller likelihood of having difficult to find bugs.

b) Consider what kind of operations will be performed on the collection's items:

- If you will mostly perform **operations over the entire set**, choose a **linked list or an un-sorted array**. An example of this is the 3D point meshes used in computer graphics which may contain millions of points, but we don't usually perform individual point look-ups; instead, we almost always render the whole mesh. Meshes are often kept in arrays.

- Conversely, if individual item look-ups (**search**), insertions, or deletions make-up the majority of the operations on your collection, then you should **choose a data structure** that gives you the **best Big O complexity** for these operations.

Note

However, keep in mind that there is **no pre-set definition** for what it means for a collection to be **small** or **large** or **very large**. So likely what you will need to do is sit down and **create a table** such as the one shown in Exercise 4.1, then run a few numbers to see what storage method will result in the best performance for the **specific collection you will be working with**, then consider **ease of implementation**. Document your findings, and then make the best choice you can **supported by having thoroughly thought through** the issues mentioned above.

4.7.2 A brief note on databases

Real world systems that manage large collections of data do not rely on **a single kind of data structure**. For example, typical **databases** organize the actual item information into **tables** - these are basically **huge arrays** (stored on disk, not computer memory), where **each row contains all the data fields for one item in the collection** (in effect, one row in a database table is equivalent to a **CDT** we may have in memory). The tables are most often un-sorted.

Separately from the table that contains the actual data, the database keeps an **index in the form of a balanced tree** (typically a **B-tree** or a variation thereof) which contains **search keys** the user may want to run queries with. **Each node in the index contains the location in the original table where the information for the item matching the query key can be found** (this is just like a **pointer**, but instead of containing a memory location, it contains a disk location).

This has several important benefits: Firstly, **the index data structure is very small compared to the size of the collection** because it only stores **keys** and **pointers to entries in database tables**. Often, the whole index can be kept in memory (whereas the entire data collection may be too big for this). This is important because working with information that is in the computer memory is much faster than working with information stored on disk. Secondly, we can define **multiple indexes** over the same collection. For instance, a database that stores information about students at a University may have an index based on **student number**, and another index based on **student name plus date of birth**. This allows the users to perform queries using either the student's name and birth date, or their student number, and both will be equally fast.

Separating the index from the data has the advantage that it allows us to provide different **ways to find information** within the collection efficiently.

The above motivates one final thought regarding the problem of managing a large collection of information: **It involves thinking at multiple levels of detail**. Knowing how particular **ADTs** and their corresponding **data structures** work, as well as the complexity of typical data operations when we use these **data structures** is only the foundation. Once you have understood that part, you can begin thinking at a higher level of abstraction: How will the collection be used? what kinds of operations will be more frequent? what kinds of look-ups may be often required and therefore should be made very fast? and how do we organize the information available into possibly many different data structures so that the overall system provides excellent performance for all required uses?

This is an extensive and fascinating topic, and you can learn a whole lot more about it by picking up a good book on databases.

4.8 Wrapping up

We have spent some time carefully studying the problem of **how to efficiently manage** large collections of information. Along the way we studied the problem of **measuring, characterizing**, and **reasoning about** the **complexity of algorithms and problems**. We have thought about **how to determine which data structure** will provide a better performance in a particular case (as described by the size of a collection and the types of operations that we will carry out on it), and we have thought about **how and when** we should use the different **ADTs** we have learned up to this point.

Keep in mind that the **theoretical analysis of complexity** is **only one layer** of the difficult problem of figuring out the most efficient way to carry out some task. You need to learn about computer architecture, how modern **CPUs** work, and how to optimize programs for maximum efficiency if you really want to write the best (most efficient, fastest) software to solve any given problem.

As for the **worst-case complexity**. The discussion above may lead you to believe you don't have to worry about it if your data structure has a good average-case complexity. Indeed, for most applications you will find you can get excellent results from using data structures such as **BSTs**. However, **for safety-critical applications**, or for **real-time applications**, examples of which include **industrial control software, medical equipment, electrical power generation, transportation (aircraft flight control), robotics, manufacturing, and so on**; you can **not afford to use a data structure whose worst-case performance is bad**. The point being that even if you need to be very unlucky to get the input that triggers worst-case or close to worst-case performance, you can not afford to take that risk. And this is due to the fact we often have to worry about malicious parties (i.e. hackers and other adversarial entities) who may use knowledge of how your system is built to find a weak point and exploit it. Applications of the type just mentioned **require guaranteed performance bounds** from all the data structures and algorithms used within their software.

4.9 Additional Exercises

✍ **Exercise 4.10** Draw the **BST** that results from inserting the following **keys** in sequence into an **initially empty BST**. Keys are sorted alphabetically.

"Iron Man 3", "Black Panther", "The Avengers", "Captain America", "Iron Man 2", "Aquaman", "Batman Returns", "Thor", "Spider Man: Into the Spiderverse"

✍ **Exercise 4.11** Draw the BST after we delete "Aquaman"

✍ **Exercise 4.12** Draw the BST after we delete "Iron Man 3"

✍ **Exercise 4.13** What is the list of movies generated by a **post-order traversal** of the **BST** from Exercise 4.12?

✍ **Exercise 4.14** It is highly likely that a **BST** whose **keys are movie titles** will run into the problem that there are multiple entries with the same key. As we discussed above this is really not a great situation. Give at least **3 different suggestions** regarding how we can build a **BST** where entries are organized by movie title, yet, there are no duplicate keys.

✍ **Exercise 4.15** Which of the following has the **lowest complexity** for large values of **N**?

- $250000 \cdot log(N)$
- $.001 \cdot N$
- $.000001 \cdot N^2$
- $5000 \cdot log(N^2)$
- $.01 \cdot N \cdot log(N)$

✍ **Exercise 4.16** Which of the following has the **lowest complexity** for small values of **N**?

- $5.25 \cdot N$
- $1.11 \cdot N$
- $5 \cdot log(N)$
- $2.1 \cdot log(N^2)$
- $.00001 \cdot N^2$

✍ **Exercise 4.17** Typically, changes to the **key values** in nodes for a **BST** are not allowed because they could break the **BST property**. However, in practice we may come upon a situation in which **we may have to update a data field used as key**. List the steps we could take to accomplish this without breaking the **BST property** in the tree. Write the process in pseudo-code, and then provide a function in **C** that does this and add it to your **BST** application to handle movie reviews.

Use the test-as-we-develop approach as described at the end of Chapter 3, documenting your tests for the new function and adding them to the **test-driver** you have been working on for the restaurant reviews app.

After your new function is thoroughly tested on its own, **add a set of tests** to verify it works well with the rest of the program, as per the guidelines in Chapter 3.

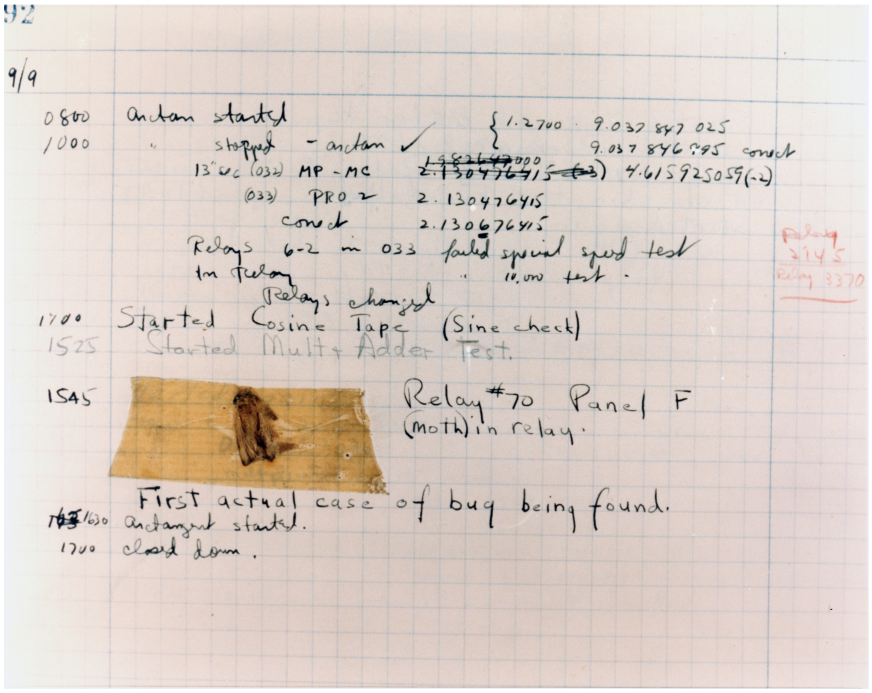
Finally, add an option to **main()** so the user has access to the new functionality. It should allow the user to change the name of the restaurant (which we are using as key) for any of the reviews **already present** in the **BST**.

✍ **Exercise 4.18** Implement **tree sort** from the description in the text (at the end of the subsection on in-order traversals). Your **tree sort** function should take as input an **un-sorted** array of integers (or floats, up to you), and return the **sorted array** using **tree sort** to carry out the sorting process.

4.10 Building programs that work - Part 4

In the previous Chapter we discussed the process for having a solid **testing strategy** that will allow us to thoroughly check that our program's functions are working as intended, and to verify that the completed software performs its task correctly on any input that may be provided to it.

A **solid testing process** will allow you to find **bugs** - errors in the way the program handles information that result in wrong behaviour. Examples of this can be output that is not correct given the input, it can be behaviour in the application that is not expected, it can be a serious problem that causes the program to break, or it can produce unexpected changes to information being managed by your program. The term **debugging** has been around for a long time (some say since the late 1800's) but you can see the first actual **bug** found inside a computer in Fig. 4.20.



92

9/9

0800 Anctan started {1.2700 9.037 847 025
1000 " stopped - anctan ✓ 9.037 846 995 conect
13" sec (032) MP - MC ~~1.982647000~~ ~~2.130476415~~ ~~(-3)~~) 4.615925059(-2)
(033) PRO 2 2.130476415
conect 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 10,000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F (moth) in relay.
First actual case of bug being found.
~~143~~ 1630 Antangent started.
1700 closed down.

Relay 2145
Relay 3370

Figure 4.20: The first bug found inside a computer. This is from 1947 and resulted in erroneous results from math computations. *Photo: U.S. Government, Public Domain*

Though the variety of bugs that you will encounter while developing software is very large, the **process for finding and correcting them is the same**. The first step is to identify the existing of a bug. This may occur in different ways:

- The program fails one or more of your tests during development. This is normal while developing software and is the best way of finding and fixing bugs.
- A user reports a problem with the software once it's been released. Which should happen much less often if you have done your work properly. We want to avoid this, it is harder to find and fix bugs that were reported by users.
- Another developer, working with your program, reports a bug. This is common when working on **open source** projects which involve a community of developers working together.

Regardless of how the bug was initially identified, the process for fixing it involves:

- Figuring out how to reproduce the bug. With a failed test this is easy, you know which test failed and can run this test anytime to **trigger** the bug. With user-reported bugs it is a bit more difficult as we often don't have enough information about what was going on when the bug occurred. With bugs identified by a developer, we usually get more information but we also have the option to contact the developer to figure out what **sequence of inputs or actions** can be used to **trigger** the bug.

- Once we have a way to **trigger the bug**, we need to determine **what is the correct behaviour** the program should have for this **test case**, **input sequence**, or **user action** (whatever the case may be) - we **can not fix a bug if we do not know what the program should do in the correct case**.
- Once we have both **a way to trigger the bug**, and **a thorough understanding of what the program should do in the correct case** it's time to **debug the program**.

4.10.1 What does it mean to debug a program?

The process of fixing a bug, **i.e. debugging**, consists of **tracing through the program** to check **each step the program carries out** to identify **at which point the program produces an unexpected/wrong result or behaviour**. Usually, in a well designed program, we start by **identifying the function** (or perhaps a small sequence of function calls) within which the bug occurs.

Once we have identified the function(s) where the bug occurs:

- We have to check what is happening at each step/line within the function(s) on the **input that triggers the bug**. We need to know what each step in the function is supposed to do, so we can verify each step and find which ones produce the wrong results.
- Once we have identified the line(s) where what the program does is different from what it should be doing:
  - Inspect the content of local variables, input arguments, arrays, and any other data that the program is using.
  - If the information the program is using is correct, then the lines of code that produce the wrong behaviour are likely wrong. Think through them carefully, trace the program on paper for the lines that are the likely problem and figure out which step(s) are incorrect, then fix them.
  - If the data the program is using is **not as expected**, then the source of the bug is somewhere else. You need to **go back to tracing the program** but this time you need to pay attention to where the data you identified as having the wrong value is being manipulated. Once you identify the section of the code where data gets modified or set incorrectly, think through that code, trace what it does on paper, and fix any problems you find.

Keep in mind that **if the problem is with information being used by a function**, the actual **origin of the bug** may be in an entirely different part of the program. However, **once you know which piece of information is getting wrong values** you can trace the program looking at parts that have access and can change that specific information to figure out where things are going wrong.

4.10.2 How do we trace a program?

The simplest way is to **add a lot of print statements**, each of which has to provide you with (at the very least):

- The specific location at which the print statement is taking place - so when you see something wrong you can immediately find what part of the program the corresponding information was printed from.
- Values of any variables, input arguments, arrays, or data structures that are being used at the program at that specific point, and which you need to know in order to determine if the program is doing the right thing at that particular point.
- If the print statement is within a loop, the **iteration of the loop**, so if a problem is detected you know at what point in the loop it happened.

Your task is then to read through the sequence of printed information, to identify where in the sequence something is not correct, and then to go into the program to determine what may be the problem at the location corresponding to where the print statement was produced.

This is a general way to trace through a program, and will work on a wide range of settings. It doesn't require special tools being available. But on the flip side, it doesn't provide you with control over the process of **tracing**. You can not **stop the program** to inspect what is happening in memory, and you have to dig through **possibly very long** sequences of printed information.

A more powerful, but also more complex alternative to using print statements is the use of a **debugger**. This is a specialized tool that allows you to control the process of **tracing through a program** so you can run each step, then have a look at what is going on in memory with the program's variables and data, and then continue with the next step once you have checked everything is ok up to that point.

A standard **debugger** is called **gdb**, and is available in all **Unix/Linux** systems, which includes a large proportion of **computing servers** and other systems you may need to work with. So knowing how to use **gdb** is a valuable skill. The **gdb debugger** is a fairly sophisticated program, so if you want to learn to use it you should spend a meaningful amount of time working through a tutorial on the subject. Many such tutorials are available both as electronic documents and in the form of how-to videos. If you want to follow up on this, here's a good place to get started: `https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf`.

The goal of using a **debugger** is the same as that of using print statements - you want to be able to **inspect what is happening with the program at each step** so that you can **check what is happening against what you know should be happening** and identify the place(s) where something is wrong.

4.10.3 Pitfalls to avoid

Whenever there is a problem with a program it is very important to remember: **computers are deterministic, predictable, and execute the instructions the program contains**. It is easy to fall into one of the following **pitfalls** by thinking that:

- The program is correct, **it must be some random thing with the computer**. This is **incorrect**. Computers are predictable and deterministic. If they do something, it is because the program caused them to do that. Believing the program is correct against the evidence that there is a problem is one sure way to fail to fix the problem.
- It works on a different computer, therefore it's a problem with the computer. This is **most often incorrect**. It can actually happen that a hardware problem may cause a program to fail in weird ways but this is **very rare**.

Most likely, there are differences in how the program is running on these computers and these differences cause the bug to trigger in one but not the other. This requires **very careful investigation** - with very high probability the program has a bug and thorough testing will find it.

- It fails because the test was **tricky** so it is ok, we don't need to fix anything. This is **incorrect**. Thorough testing **requires** that we **try to break the program**. This means testing on **unexpected** or **possibly incomplete or incorrect** inputs and verifying that the program does the right thing and doesn't fail. Our programs should **not fail** because we have not devoted enough thought to making them solid.
- The bug is probably very difficult to trigger by a real user (i.e. it would show very very rarely) so it is ok not to fix it. This is also **incorrect** and **unethical**. If we know there is a problem with the code, **it is our duty to fix it**. Something like this: `https://www.cnn.com/2019/05/05/us/boeing-737-max-disagree-alert/index.html` is bf wrong and should never be accepted.

Some bugs will be much harder to find and fix than others, and the more complex the software is, the more time and effort it will likely take to test and debug thoroughly. However, if you are careful and follow a thorough, logical process to testing and debugging you can **find and fix** any reported bug. **Avoid convincing yourself that it is acceptable to ignore a problem**.

4.11 Time to Practice

In order to practice the process we have established for developing working software, from program design, to testing, to debugging, we will look at one fairly short example that nevertheless will allow us to practice all of the ideas we have covered thus far.

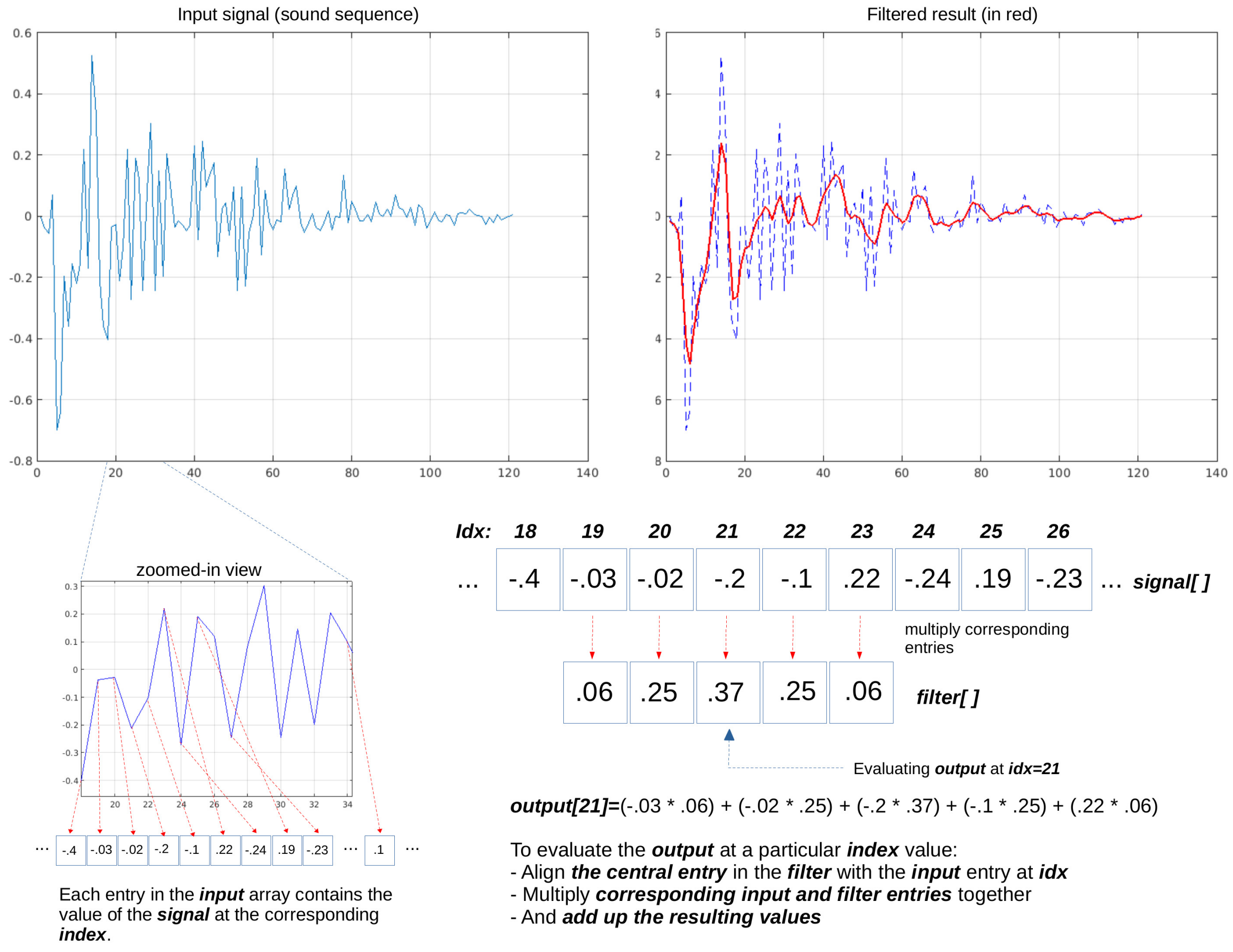
The problem: Our task is to implement a **signal filtering** function. Signal filtering is a common task in applications that deal with **media**, **communications**, and **machine learning** among others. A common example of filtering is **smoothing a signal**, for instance in order to reduce noise. Typically, we will have access to **an input** which usually represents a signal of interest, such as an audio recording; and a **filter** which is a **pattern of values** we will use to combine **data in the input** to produce a result. The filtering process is illustrated in Fig. 4.21.

The **filter** must be **odd-length** since we have to align its central entry with each input entry in order to compute the corresponding output. Normally, the length of the **filter** is much smaller than the length of the signal. For example one second of high-quality sounds data would contain over **40,000** entries, whereas a typical **smoothing filter** would have 5, 7, or 9 entries.

The process itself is straightforward, as shown in the Figure.

4.11.1 Step 1 - Understanding the problem and designing the solution

As discussed at the end of Chapter 2, the first step in implementing a program that works for solving the above problem is to **understand the problem** thoroughly. One way to verify you've understood the process above, is to do an example of the filtering process on paper, with a short filter and maybe one or two different input locations. You can use this later on when designing the tests for functions in the program.


Input signal (sound sequence)
Filtered result (in red)
zoomed-in view
Idx: 18 19 20 21 22 23 24 25 26
... -.4 -.03 -.02 -.2 -.1 .22 -.24 .19 -.23 ... signal[ ]
multiply corresponding entries
.06 .25 .37 .25 .06 filter[ ]
Evaluating output at idx=21
... -.4 -.03 -.02 -.2 -.1 .22 -.24 .19 -.23 ... .1 ...

Each entry in the ***input*** array contains the value of the ***signal*** at the corresponding ***index***.

output[21]=(-.03 * .06) + (-.02 * .25) + (-.2 * .37) + (-.1 * .25) + (.22 * .06)

To evaluate the ***output*** at a particular ***index*** value:
- Align ***the central entry*** in the ***filter*** with the ***input*** entry at ***idx***
- Multiply ***corresponding input and filter entries*** together
- And ***add up the resulting values***

Figure 4.21: The process of filtering a 1D signal. The value of **output[i]** is the result of aligning the center of the **filter** with the entry at **input[i]**, and then adding up the result of mutiplying corresponding values. This has to be done at every **valid** entry in **input**. **Valid entries** are those for which after aligning the filter with the input, there is one input entry to multiply with each filter entry. For simplicity, the **output** value will be set to **zero** for **non-valid** entries.

✍ **Exercise 4.19** Carefully review the description of the problem in the figure, and make notes to yourself that clearly describe the process of filtering a 1D signal, including any conditions placed on the inputs and/or output of the filtering process. Work out at least one short example on paper showing the computation that has to be carried out by the filtering program.

The next step in our process is to come up with a written description of the algorithm, in enough detail that it can be implemented.

✍ **Exercise 4.20** **Before continuing,** write a complete description of the algorithm for 1D filtering. You can do this as a list of steps in bullet point format, or as pseudo code. **Do not attempt to write C code at this point**.

You can compare your description of the algorithm with the **pseudocode** below (take special note of any steps that are different/missing and if needed review the algorithm description to resolve any differences).

```
Algorithm: Filter1D - performs 1D filtering of an <input> array
                      with the specified <filter>.
                      Produces an <output> array with the same
                      length as the <input>, where each entry is
                      the result of applying the <filter> to the
                      corresponding entries in the <input> array.

Procedure:

* Loop over entries in the <input> array, one by one, using index <i>

   - For each value of <i>
      * Align the central entry in <filter> with the <input> entry <i>
      * Check that the every entry in <filter> has a corresponding
        entry in <input>
          - if this is not the case, set <output[i] := 0> and
            continue with the next <i>
      * Compute the result of applying the <filter> to the
        <input> at <i>: Multiply corresponding entries, and
        add up the result of these products (this operation is
        called a cross-correlation)
      * Update <output[i]> to be the result of the cross-correlation
```

Having understood the problem, we have to design our solution. There is **always more than one way, and often many ways** to do this, and in most cases **there is no single correct way** to solve any reasonably interesting problem. So your focus here is not **to find the one right solution** but rather to design a solution that **makes sense to you**, follows **good design practice** in terms of breaking up your algorithm into **self-contained functions**, that you can **test and debug** with reasonable ease, and that, where applicable, **uses data structures and algorithms with good known complexity** for the problem at hand.

✍ **Exercise 4.21** **Before continuing,** come up with your own design for a program that implements the pseudocode shown above. This means deciding how to break the problem into functions, and figuring out what the information flow will be between these functions and **main()**.

In the case above, a reasonable way to break up the problem into functions we have to implement could involve:

- A function called **crossCorrelation()** that computes the **cross-correlation** between an **input** array and a **filter** array at a specified index **i**
- A function called **filter1D()** that **loops over the input array** calling the **cross-correlation** function to obtain the value of the **output**

As noted above, this is only one way to solve the problem, someone else may decide to do everything in the algorithm within as single function. Both of these solutions have their own advantages and disadvantages. We will proceed with the breakdown above.

4.11.2 Step 2 - Testing as we develop

At the end of Chapter 3, we looked at the process of **testing a program as we are developing it**. This involves **choosing the order of implementation** for the various functions, then **implementing them in order** and **thoroughly testing each function** as we complete it before moving on to the next one.

For the problem at hand, we have two functions, and it is fairly clear that **cross-correlation** has to be implemented first because it is used by the function that computes the **output** array. Therefore, we will

- Implement **crossCorrelation()**
- Thoroughly test **crossCorrelation()**
- Implement **filter1D()**
- Thoroughly test **filter1D()**

and once we have completed the steps above, we will then **thoroughly test the complete process** of filtering 1D signals to make sure it works for a variety of possible inputs as well as a variety of filters.

✍ **Exercise 4.22** Implement **crossCorrelation()** and **filter1D()** following the design and pseudocode described above. There is a reference implementation below, which you will use for **testing** and **debugging**, but you should first implement these functions yourself - compare your implementation with the reference below and carefully consider any differences. You can apply the tests and debugging process described below to your own implementation as well as the reference one.

✍ **Exercise 4.23** Write down a set of tests for the **crossCorrelation()** function, the **filter1D()** function, and **the complete working code**. The tests should be thorough and allow you to discover any potential bugs or errors in the computation being performed by the filter.

4.11.2.1 Step 3 - Debugging

In order to practice **debugging**, consider the implementation below for the two functions that comprise our 1D filter:

```
double crossCorrelation(double input[],int n, double filter[], int m, int idx)
{
  // This function returns the cross-correlation between an input
  // array and a filter array at input location 'idx'.
  //
  // Inputs: input[] - a 1D array of size n
  //         filter[] - a 1D array of size m
  //
  // n>m , and m must be an odd integer number
  // (n-1)-hs >= idx >= hs
  //
  // The cross correlation is defined as
  //
  // cross_correlation=sum_{i = -hs}^(hs) input[idx-i]*filter[i+hs]
  //

  double cross_correlation=0;
  int hs=(m-1)/2;               // Half the filter size
  int i;

  for (i=-hs; i<hs; i++);
  {
    cross_correlation+=input[idx-i]*filter[hs-i];
  }

  return cross_correlation;
}

void filter1d(double input[], int n, double filter[], int m, double output[])
{
 // This function takes a 1D input array of size n, and a 1D filter of
 // size m, and produces an output result (same size as the input) that
 // is the result of evaluating the cross-correlation between the input
 // and the filter at each valid location.
 //
 // Inputs: input[] - a 1D array of size n
 //         filter[] - a 1D array of size m, m<n, m is odd
 //         output[] - a 1D array of size n
 //

 double cc;
 int hs;
 int i;

 hs=(m-1)/2;

 for (i=0;i<=n;i++)    // Initialize output array to zeros
   output[i]=0;

 for (i=0; i<=n; i++)
   output[i]=crossCorrelation(input,n,filter,m,i);
}
```

The code above looks reasonable and correct, but it in fact **contains bugs**. Your task now is to use the tests you developed above to **identify problems** with the implementation, and once you have found a problem, **debug it** by following the process described in this Chapter.

✍ **Exercise 4.24** Use the set of tests you developed to find **bugs** in the implementation above. Once you identify a **bug**, carefully **trace through the code** using either **carefully placed, informative printf() statements**, or a debugger such as **gdb** in order to identify where the problem is, and then correct it. Remember you must be able to follow each instruction in the implementation, and you must know what the correct result of each instruction must be (in terms of the variables and data involved in them).

If you do not discover any problems with your tests, that would indicate your set of tests was not thorough enough, and you should revisit Chapter 3 and the process of testing, then expand or change your set of tests accordingly. The end result of working your way through this section and completing all of its exercises as well as the debugging task should be an implementation that is thoroughly tested, solid, and that will perform 1D filtering on any input signal and **valid** filter correctly.

✍ **Exercise 4.25** Carry out the testing and debugging process on your own implementation of the functions for the 1D filter. Once you are satisfied that your own implementation works correctly, perform an additional set of **high-level tests** that consist of filtering multiple, different input signals with various filters, using both the (now fixed) reference implementation, as well as your own. **compare the output** from both of these for each input/filter test case, and ensure they are identical for every test. If you find any differences, then either or both of the implementations still contain bugs. Find and correct any problems found in this manner.

The final exercise in this section illustrates one more way in which we can **test** a program we have developed: by **comparing it** with a different implementation (which can be in a different programming language, or simply written by a different developer or team). This is common practice for software that implements functionality that is well known or common, and for which we can easily find reference implementations available for our testing. However, be careful:

- **Do not copy or immitate code from the reference library** - Code that has been developed by someone else is protected by copyright, copying it or using trivial variations of it is an infraction. You are expected to develop your own solution independently and without looking at the work of others. We will discuss how to **incorporate and re-use** code from **open source** or **creative-commons** libraries in a later Chapter.
- **Do not assume** that the reference implementation is **free of bugs**. If you find a disagreement between results obtained with your own code, and that of the reference implementation, all that you can conclude from this is that **at least one of them has a bug**, you have to carefully trace through your program, and if necessary also the reference implementation in order to determine which one(s) have errors.

Chapter 5 Graphs and Recursion

At this point, we have a pretty good handle on how to **store, organize, and access data**. We have learned about **computational complexity**, and how we can use complexity analysis to gain a better understanding of different problems, and of the algorithms we implement to solve them. We have in our toolkit a couple of very useful **ADTs**, and we have spent a good amount of time practicing how and where to use them.

In this Chapter, we will add to our toolkit two of the most general and powerful tools for problem solving in computer science: **Graphs**, and **recursion**.

Graphs are used to model all kinds of real-world items, and real world problems, where the key to understanding the particular problem is the **relationship between items** in our collection. **Recursion** gives us the ability to understand, manipulate, and solve problems whose particular properties make regular processing with loops and conditionals very difficult.

Together, **graphs** and **recursion** will open the door for you to work on a variety of fascinating applications in all fields of knowledge. So let's dive in and find out what we can do with these two tools.

5.1 Graphs

In computer science, **graphs** are used as a model to represent items of interest, where the **relationship between items is relevant** to the problem we wish to study or solve. To understand this, let's take a look at the sort of information we have been working with up to this point, and the kinds of problems we have been solving with the tools we have acquired in previous Chapters.

Thus far, we can work with (possibly very large) collections of data items, these items can be regular **C** types, or **CDTs** which contain multiple fields. However, one key property of the data we have been working with, and the problems we have been looking at, is that **each item is fundamentally independent** from the rest.

For example, we have been working with restaurant reviews - we can search for specific restaurants, find out which restaurants have a review score above a specific value, check out the restaurant addresses, and so on. Importantly: Each review is processed independently of the rest, and the problems we have been solving do not require us to model in any way the possible relationships between reviews for different restaurants.

However, **relationships between data items are extremely important**. In the case of our restaurant reviews, for instance, we may want to consider that different locations of the same food chain are related to each other: They serve the same food, so we should expect their scores to be similar to a large degree. Restaurants offering a particular type of food (e.g. Mexican tacos) are related, and comparing their reviews is informative. Restaurants in the same part of town are also related (geographically), and this is an additional source of information that we haven't used thus far: Perhaps users in a particular part of town are more *picky* and like to give worse reviews regardless of how good a restaurant is, or perhaps they just enjoy certain kinds of food more than others. The tools we have available up to this point do not allow us to study any of these meaningful connections.

Graphs provide us with a way to **model, reason about, and manipulate data items and the relationships between them**. With **graphs**, we can ask questions such as *what kinds of restaurants do my friends like?*, *what are the best courses people I know take in the 2nd year?*, *How good are movies directed by Martin Scorsese?*, and *What* *is the fastest route to get from my home to the wonderful pizza place that I love best?*

Graphs store **data items**, and their **relationships or interactions**, in a way that allows us to implement algorithms that explore the structure of the data we are studying, and because of that, they allow us to **find meaningful interactions** between data items, and to **look for interesting patterns** in complex data sets. Because of this, **graphs** are central to a large variety of methods in machine learning, artificial intelligence, and data visualization. They are used to represent information that can be conceptualized in the form of a **network** - that is, **nodes** and their **connections**. We already know what a **node** is in terms of data representation and storage, let us look at the **connections** and what they may mean. Typical applications of **graphs** include:

- Social networks - where **nodes** represent people, and **connections** join pairs of people who interact socially in some particular context.
- Transportation - where **nodes** represent locations, and **connections** join pairs of locations that we can travel between (for example, two intersections joined by a street). This also applies to **internet routing**, **electrical or water distribution** grids, and other similar problems.
- Genomics and bio informatics - where **nodes** can represent things like proteins, DNA sequences, enzymes, etc., and **connections** join together pairs of these that interact within a biological system we are modelling.
- Courses and their pre-requisites - where **nodes** represent courses, and **connections** represent the pre-requisite structure (if there is one) for pairs of courses to be taken.
- Databases - where **nodes** represent data tables (we briefly read about them in the previous Chapter), and **connections** represent the structure of the data in the Database. For example, one table may contain information about a company's employees, a separate table may contain information about payroll, and the connection between them indicates that the payroll table provides information about the salaries of employees in the employee table.
- Recommendation systems, which suggest items of interest based on a user's tastes - **nodes** represent items of interest (e.g. movies, books, music, or products in an online store), and **connections** represent **similarity** so that the system can decide which items to recommend for a particular user.

The above is just a small sample of the wide range of applications for **graphs**. Because they can be conceptualized as **networks**, and because they represent **relationships** between data items, **graphs** can often be **visualized** by plotting **nodes** in some particular pattern, and showing the **connections** between them - when designed carefully, such visualizations can help humans get an overview of the data they are working with at a glance, and to visually discover interesting patterns that may be worth exploring with software. Examples of graph visualizations are shown in Fig. 5.1.

5.1.1 Definition of a graph

A **graph** consists of:

- A set V of **nodes** corresponding to data items we are working with. In books, lecture notes, and future courses, you may find the term **vertex** is used instead of **node**, they are the same thing. The size of V (the number of **nodes** in the **graph**) is N.

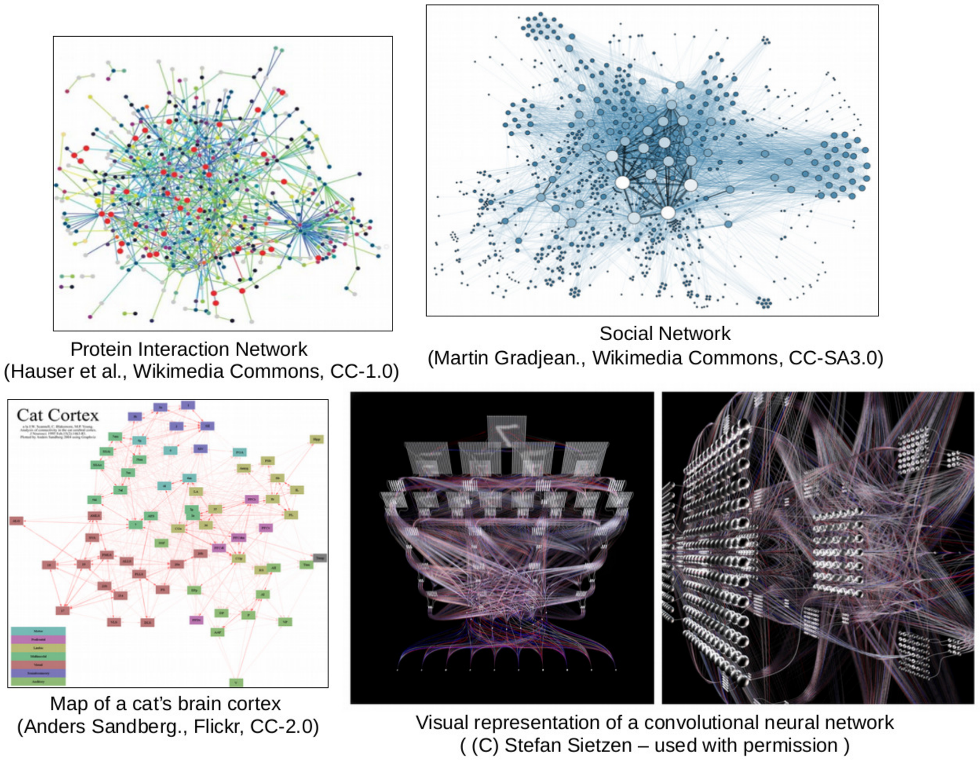

Protein Interaction Network
(Hauser et al., Wikimedia Commons, CC-1.0)
Social Network
(Martin Gradjean., Wikimedia Commons, CC-SA3.0)
Cat Cortex
Map of a cat's brain cortex
(Anders Sandberg., Flickr, CC-2.0)
Visual representation of a convolutional neural network
((C) Stefan Sietzen – used with permission)

Figure 5.1: Sample visualizations of **graphs** in different domains.

- A set E of **edges** which are the connections between nodes. They represent the relationships existing between data items in our collection. The size of E (the number of **edges** in the **graph**) is M.

Together, they define the graph $G = (V, E)$. **Graphs** are a very general way of representing information and relationships between items. You can build a graph for pretty much any problem you can think of, as long as you can find some meaningful way in which data items for that particular problem relate to each other.

We have already been working with **graphs**! Trees, such as **BSTs** are **graphs** (the nodes in the tree are related to each other by parent-child relationships), **linked-lists** are also **graphs** (nodes are related to each other by a predecessor-successor relationship). So in fact, you already have plenty of experience working with information stored in graphs.

5.1.2 Types of Graphs

There are two general types of graphs we will use widely:

- **Un-directed graphs**: Edges between nodes are shared, and the relationship between the nodes goes both ways. An example of an **un-directed** graph is shown in Fig. 5.2.
- **Directed graphs**: In this type of **graph**, edges have a direction, the relationship between the nodes goes one way. Examples of these are trees like the **BSTs** you've been working with: Edges in **BSTs** go from parent to child. If **node a** is a parent to **node b**, the reverse can not be true. An example of a directed graph is shown in Fig. 5.3.

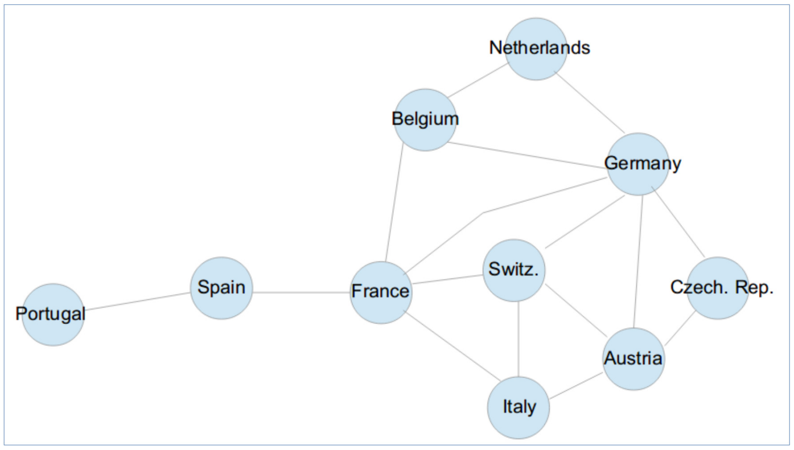


Figure 5.2: Un-directed graph representing countries in Europe. Edges in this graph indicate the corresponding countries have a shared land border (this relationship goes both ways by definition).

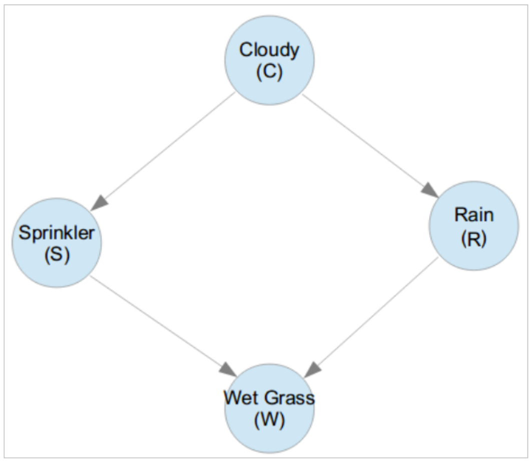


Figure 5.3: Directed graph representing binary (TRUE/FALSE) variables in an **inference problem**. The direction of the edges (represented by arrows) indicates which variables have a direct effect on each other. For instance, the **Rain** variable (indicating whether or not it has been raining) can directly affect the value of the **Wet Grass** variable. The converse is not true - wet grass can not direcly cause rain to happen.

What type of graph we use depends on the data we are working with, and the problem we are trying to solve.

5.1.3 Terminology for graphs

The following are definitions of important terms related to **graphs** that are often used in studying graph-based algorithms:

- **Neighbours:** For **un-directed graphs**, a **node v** is a neighbour of **node u** if there exists an edge joining both nodes - we say the two nodes are **adjacent**. For **directed graphs**, a **node v** is an **out-neighbour** of **node u** if there is an edge **from u to v**. Conversely, **v** is an **in-neighbour** of **u** if there is an edge **from v to u.**
- **Neighbourhood:** For **un-directed graphs**, the **neighbourhood** of **node u** is the set of all nodes that are neighbours of **u**. For **directed graphs**, there is an **out-neighbourhood** and an **in-neighbourhood**, corresponding to the sets of **out-neighbours** and **in-neighbours** respectively.
- **Degree:** For **un-directed graphs** the **degree** of a **node u** is the size (number of nodes) in the neighbourhood of **u**. For **directed graphs** we have an equivalent **out-degree**, and **in-degree**.
- **Path:** A **path** through a **graph** is a sequence of consecutive nodes that can be visited by following existing edges between pairs of connected nodes. Figure 5.4 shows an example: there is a path from Portugal to Germany that visits in sequence Portugal $\rightarrow$ Spain $\rightarrow$ France $\rightarrow$ Italy $\rightarrow$ Austria $\rightarrow$ Germany. (Incidentally, that would probably be an amazing trip to make!)
- **Cycles:** For **un-directed graphs**, a **cycle** is a **path** with at least 3 nodes that starts and ends at the same node. For **directed graphs**, a cycle is a path that begins and ends at the same node, but in this case the path can have any number of nodes.

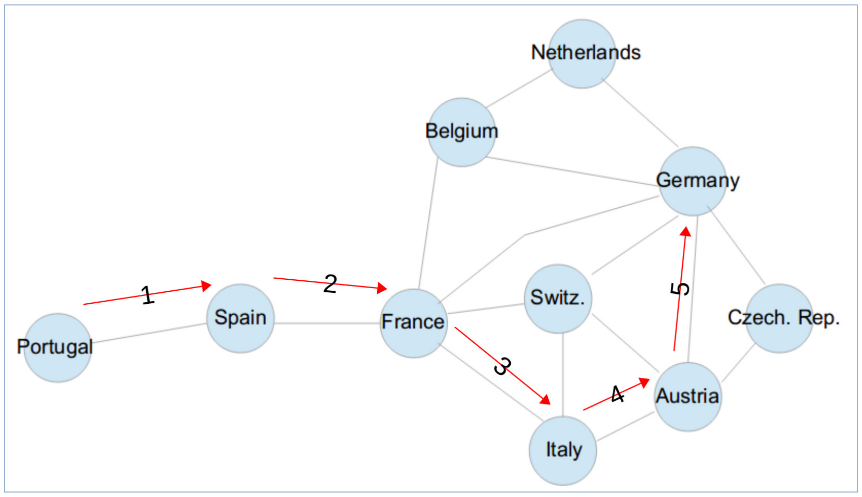


Figure 5.4: Example of a **path** in a **un-directed** graph. The **path** is the sequence of nodes visited while going from one **node** to **another**.

Note

For **un-directed graphs**, we can travel along edges in either direction while forming a path. But for **directed graphs**, we can only go from **node u** to **node v** if an edge exists that goes from **u** to **v**. This is very much like travelling in a city with lots of one-way streets, we must follow the direction of the streets while going from one place to another.

✍ **Exercise 5.1** For the graph in Fig. 5.2, what is the neighbourhood for the **Germany** node? What is the degree of this **node**? which **node** has the largest **degree**?, what **node** has the smallest **degree**?

✍ **Exercise 5.2** For the graph in Fig. 5.3, what is the **out-neighbourhood** of the **node** for **Cloudy**? What is the **out-degree** for this node? Which node has the largest **in-degree**? Is there a path from **Wet Grass** to **Cloudy**?

Note

A **graph** is an **ADT**. As we know, an **ADT** describes a way of organizing information, as well as a list of operations that must be supported by the **ADT**. In the case of **graphs**, the operations that must be supported include:

- Adding a **node**
- Removing a **node**
- Adding an **edge**
- Removing an **edge**
- Edge query (finding whether or not there is an **edge** joining two specific **nodes**) - this is sometimes called **adjacent**

In addition to the above, specific **graph** types may provide additional operations, such as finding the **neighbourhood** or the **degree** of a **node**.

5.1.4 Example applications of graphs

1) Network modeling: Including the Internet, social networks, professional networks, the electricity grid, a city's network of streets, protein interaction networks, transportation networks, etc. **Graphs** can be used in such networks for: Analysis of structure, simulation, detection of points of failure, route planning, determining the relative importance of individual nodes, and many other applications. A visual example of one such network is shown in Fig. 5.5.

2) Document analysis: Representing a collection of documents. These can be text, such as articles, books, on-line blogs, tweets, etc.), or any other kind of media such as images, music and sound recordings, video, etc. Documents can be **connected** in a number of ways, for instance, they can be linked by source (who generated the document), by type (e.g. linking together newspaper articles, separately from book chapters, journal papers, etc.), by topic (for instance, linking together all music videos of classical music), or by any other property or combination

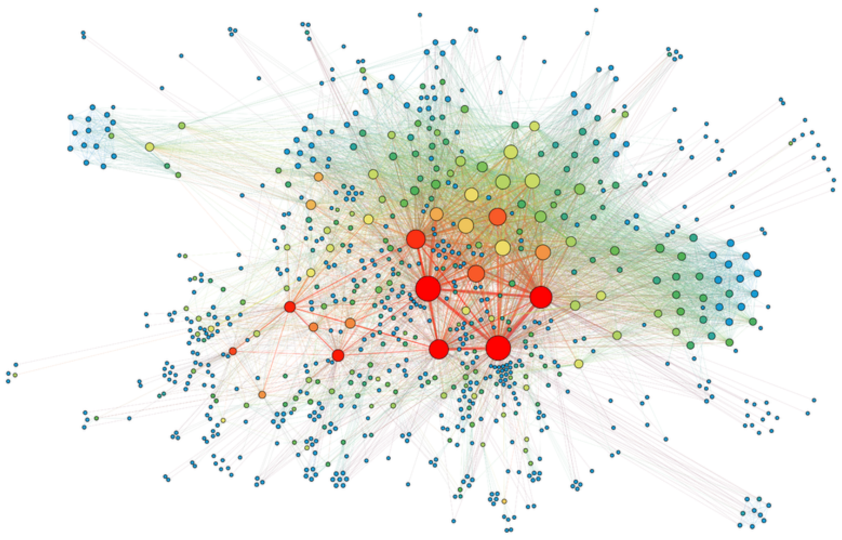


Figure 5.5: Social network analysis. Each circle is a **node** representing a person, the colour and size of the **node** indicate how **important** each node is in the **graph** - this is related to how well connected the node is, and whether its neighbours are themselves important.

of properties that is relevant to the problem we are studying. Once the **graph** has been created, we can use it to: cluster documents (group them by a relevant property), determine relevance (which documents are more commonly accessed or referenced), and discover structure in the collection. This forms the basis of recommendation systems used to determine what a user is likely to be interested in. An example of a document clustering graph is shown in Fig. 5.6.

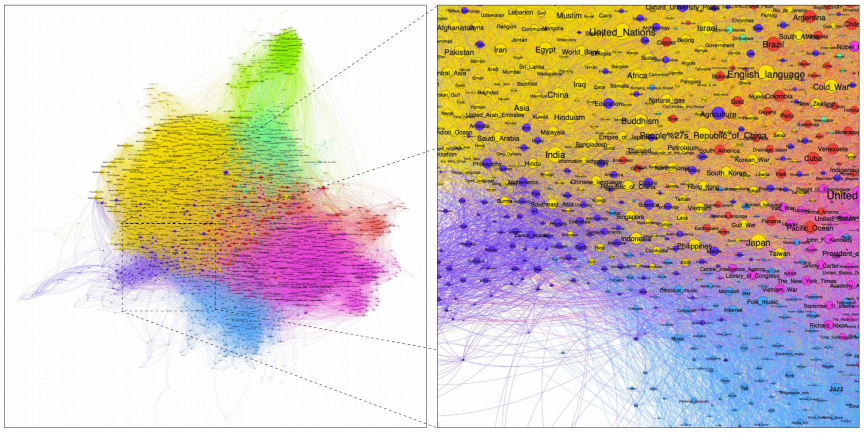


Figure 5.6: Top 2500 Wikipedia pages, grouped by similarity of content. Note that colour corresponds loosely to topic. *Image by Matt Biddulph, Flickr, CC-SA2.0.*

3) Numerical simulation: Many applications in physics, engineering, medicine, weather analysis, computer aided design, and computer graphics rely on numerical simulations performed on a mesh decomposition of a surface or volume - that is, **nodes** are placed at pre-defined locations on the surface or inside the volume, and then these **nodes** are linked to form a mesh. The values of interest for the simulation (e.g. wind speed in a weather model for a storm) are computed at each **node**, and information is propagated via the **edges** linking neighbouring nodes to carry out the desired simulation. Fig. 5.7 shows a visualization of turbulence patterns for airplane modelling.

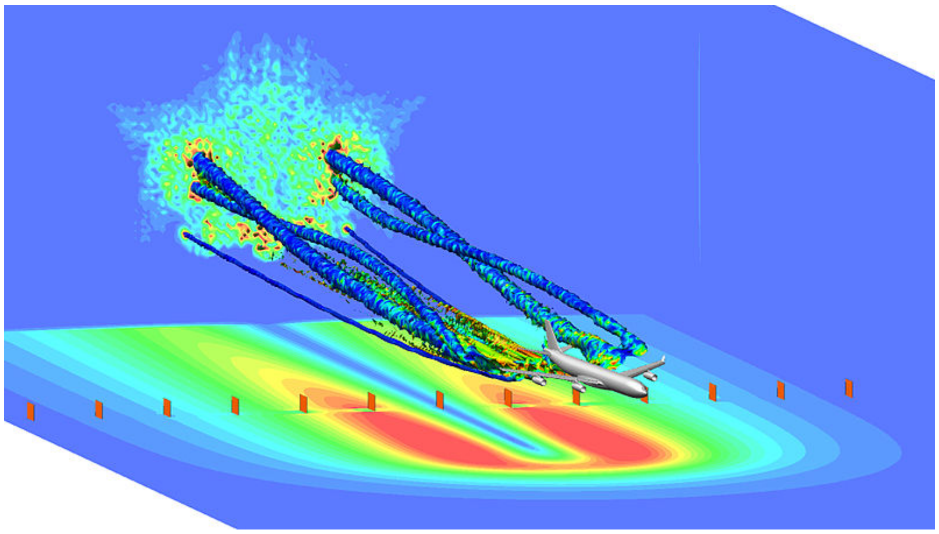


Figure 5.7: Simulation of the turbulence generated by an A340. A mesh of nodes (not visible in the image) distributed over the volume of this simulation forms the basis of the computation. The same mesh is used for visualization, by assigning a colour to nodes of interest. *Image: Deutsches Zentrum für Luft und Raumfahrt, Wikimedia Commons, CC-By3.0.*

4) Artificial Intelligence: A significant number of important applications in AI (and hence in Machine Learning) rely on graphs for representing information and for carrying out relevant processing. A very small sample of the kind of problems you can solve with graphs in AI include: path planning and route-finding, constraint satisfaction (scheduling and industrial process optimization), game playing (this has serious applications in finance, advertising, etc.), inference and decision making under uncertainty, and the current and very promising field of deep learning and its applications. One such application is illustrated in Fig. 5.8 in the context of path finding for an aerial robot.

It should be clear to you that **graphs** have an amazingly wide range of applications, and there is a variety of courses in computer science and other scientific disciplines in which you can learn as much as you like about particular problems you are interested in. In this book, our goal will be to understand the fundamental concepts related to **storing, manipulating**, and **using graphs** defined over collections of data. What you learn here will be the basis for later understanding specialized material in almost all areas of application of computer science.

5.2 Representing graphs

There are several different ways of representing **graphs** (both un-directed and directed). They each have their own advantages and disadvantages, and the choice of method for any particular application will depend on how the graph will be used and must include a careful analysis of the **complexity** of performing the tasks the graph is meant to support. In what follows, we will consider two of the most often used methods for representing and managing **graphs**. Each method must solve two problems:

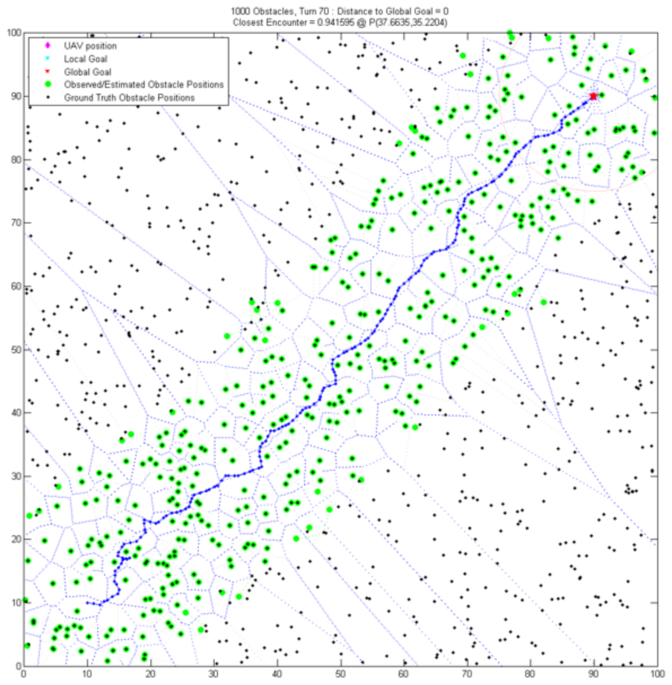


Figure 5.8: A path-planning simulation for an autonomous flying vehicle carrying a search and rescue mission in a forest. *Image: Elucidation, Wikimedia Commons, CC-SA3.0.*

First, storing and managing the set V of **nodes** that correspond to the data items in our problem. These can be stored using any of the data structures we have studied up to this point (e.g. arrays, linked lists, trees, etc.). The choice must be considered carefully, and has to be based on what **operations** will be performed on these data items, and the **complexity** of these operations.

That leaves the problem of storing the set E of **edges** for the **graph**. We need a way to keep track of which **nodes** are connected, and in the case of directed graphs, the direction of each edge. The two methods we will discuss below deal with this particular problem in different ways - which will affect the **complexity** of performing all supported **graph operations**, as well as the **amount of space** required to store the edge information.

1) Adjacency List: The **adjacency list** is an **array with one entry per node**. The i^{th} entry in the array contains a **pointer to a linked list** that stores the **indexes** of **nodes** to which node **i** is connected. This is illustrated in Fig. 5.9. Adjacency lists have the advantage of being **space efficient** - if the **graph** has a large number of nodes, but each **node** is connected to at most a few neighbours, then the **adjacency list** stores the required edge information in a very compact format - without wasting memory. Conversely, common graph operations such as the **edge query** require list traversal, which as we know can be slow.

2) Adjacency Matrix: As the name implies, the **adjacency matrix** is a **2D array** of size $N \times N$, For **un-directed graphs**, entries $A[i][j]$ and $A[j][i]$ are both **1** if there is an **edge** joining **node i** and **node j**; it is 0 otherwise. For **directed graphs**, entry $A[i][j]$ is set to **1** if there is an edge **from i to j**, and is **0** otherwise. This is illustrated in Fig. 5.10. Note that for **un-directed graphs**, the **adjacency matrix** is **symmetric**.

Adjacency matrices have the same advantages and disadvantages of arrays: Edge queries are fast (no traversal required). Adding or deleting **edges**, and finding out whether two **nodes** are connected requires a single access to

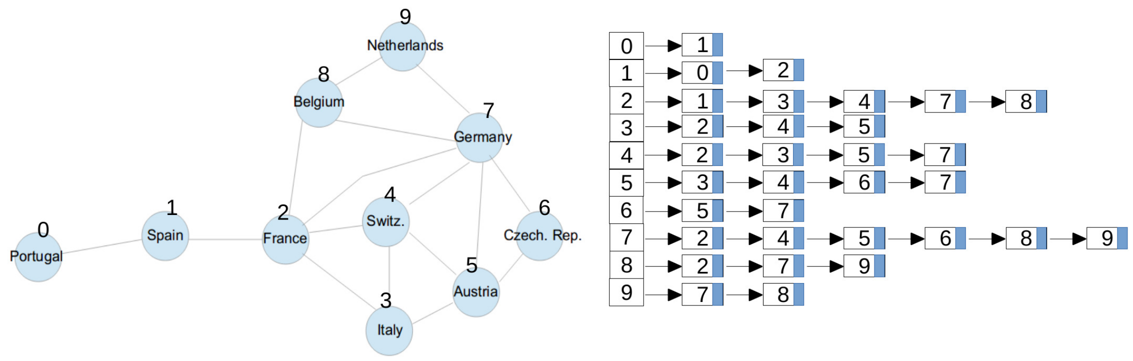
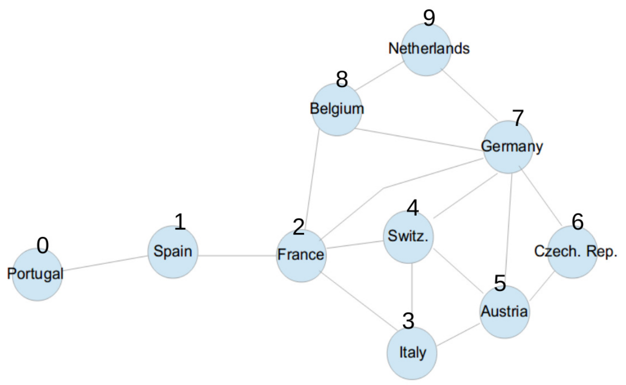


Figure 5.9: An adjacency list for the European countries **graph**. There is one entry per **node**, and each entry points to a linked list containing the indexes of the neighbours for that node. Therefore each entry in this linked list represents an **edge** in the graph.

the matrix. Conversely, they are not space efficient. Even in the small example in Fig. 5.10, you can see that the majority of the entries in the matrix are zero. For a very large **graph**, the **adjacency matrix** will waste a significant amount of space and may in fact not fit within the available memory.



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Figure 5.10: An adjacency matrix for the European countries **graph**. An edge is indicated by an entry in the matrix whose value is **1**, for example, $A[0][1]$ and $A[1][0]$ are set to **1** to account for the edge joining Portugal and Spain.

Note

For general **graph** applications, each **edge** may be associated with a **value** that represents **relevant information** about the relationship between the two bf nodes. For example in a **graph** that describes the **similarity** of songs, the **edge value** may represent **how similar** two songs are, with higher values indicating greater similarity and low values indicating the opposite.
If the **graph** has **weighted edges**, the **edge values** have to be stored. If using an **adjacency list**, each entry in the list will store **the index** of the connected **node** as well as the **edge value**. If using an **adjacency matrix** we simply store the **edge value** in the corresponding entry of the matrix. In this latter case, the **edge value** can not be zero (which indicates no connection).

✍ **Exercise 5.3** For the small example graph shown in Fig. 5.11:

- Show the **adjacency list**, use indexes $0-6$ corresponding to the nodes labeled $A-F$
- Show the equivalent **adjacency matrix** for this graph

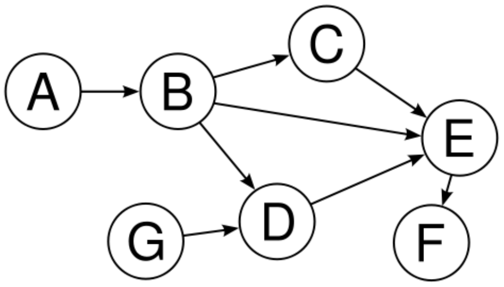


Figure 5.11: A small directed graph. *Image: David W., Wikimedia Commons, Public Domain.*

5.3 Complexity of Graph Operations

Let us consider the complexity of each of the operations the **graph ADT** specifies must be supported in the context of the two strategies discussed above for storing the **edge** information.

1) Adding an edge to the graph. Adding an edge to connect **node i** to **node j** has a **worst case complexity** of

- $O(1)$ for an **adjacency list** - we need to go to the entry for **node i**, and insert index **j** in the linked list found there. We can insert this index **at the head of the list** so the cost is $O(1)$ - if the **graph** is **un-directed** we also have to add index **i** to the linked list for **node j**.
- $O(1)$ for an **adjacency matrix** - we need to set entry $A[i][j]$ to **1**, with a cost of $O(1)$. For an **un-directed graph** we also have to set entry $A[j][i]$ to **1**.

2) Removing an edge from the graph. Removing an edge connecting **node i** with **node j** has a **worst case complexity** of

- $O(N)$ for an **adjacency list** - we need to go to the entry for **node i**, and there we will need to **traverse the linked list** until we find the entry for **node j** which we will remove. The linked list for **node i** will have a length equal to **degree(i)** (the number of neighbours of **node i**), which in the worst case is $N-1$ (the node is connected to every other node in the **graph**). For an **un-directed graph** we have to repeat the process with the linked list for **node j** to remove the entry for **i**.
- $O(1)$ for an **adjacency matrix** - we have to set entry $A[i][j]$ to **zero** (and also entry $A[j][i]$ to **zero** if the **graph is un-directed**). These operations have a cost of $O(1)$ for the 2D array that stores the matrix.

3) Adding a node to the graph. Adding a new node has a **worst case complexity** of

- $O(N)$ for an **adjacency list** - the list is an **array with size N**, so to add one node we have to **create a new array of size N+1** then we have to copy the N entries in the original array to the new one.
- $O(N^2)$ for an **adjacency matrix** - the matrix is an $N \times N$ array, so to add a node we have to **create a new array of size** $N+1 \times N+1$ and then copy the $N \times N$ entries from the original matrix to the new one. This is a computationally expensive operation for large graphs, and we also have to consider that we will need at least $2 \times N^2$ memory space to carry out the copying process, which may not be feasible. For this

reason **adjacency matrices** are appropriate only for cases in which the **graph** isn't changing frequently, or for smaller graphs.

4) Removing a node from the graph. This has **worst case complexity** of

- $O(M)$ for an **adjacency list** - to remove a **node**, we have to remove all **edges to and from this node** in the **adjacency list**. This requires traversing all linked lists of edges and removing any that correspond to the **node we are removing**. There are M entries in these lists for a **directed graph**, or $2M$ entries for **un-directed graphs**. In the worst case, for a **fully connected graph**, $M = N^2$ so it is also correct to state that the **worst-case complexity** of removing a node when using an **adjacency list** is $O(N^2)$.
- $O(N)$ for an **adjacency matrix** - removing all edges **to and from a node** requires us to set to **zero** all the entries in the **row** and **column** corresponding to that node. This involves visiting $2N$ entries in the **adjacency matrix**.

Note

In the above, the actual **adjacency matrix** or **adjacency list** are not resized - this is the most common way to implement **node** removal because it allows all other existing nodes to **keep their current index**. Remember that the actual **data items** for each of the nodes in the **graph** are stored in a separate data structure - and the **node indexes** connect our **adjacency list** or **adjacency matrix** to entries in this data structure so we have access to the relevant data when we need it.

5) Edge query. Figuring out whether there is an edge connecting **node i** to **node j** has a **worst case complexity** of

- $O(N)$ for an **adjacency list** - we have to **traverse the linked list** for **node i** and see if we can find an entry for **j** in this list. The list has a length of $degree(i)$ (the number of neighbours of **i**). I the worst case, for a fully-connected graph, $degree(i) = N - 1$.
- $O(1)$ for an **adjacency matrix** - we can directly check the value of entry $A[i][j]$ in the **adjacency matrix** and see if it is non-zero.

✍ **Exercise 5.4** Figure out and explain the **worst case complexity** of finding the **degree of node i** ($degree(i)$) using an **adjacency list** and using an **adjacency matrix**. This is for an **un-directed graph**.

✍ **Exercise 5.5** Figure out and explain the **worst case complexity** of finding the **in-degree of node i** ($in - degree(i)$) using an **adjacency list** and using an **adjacency matrix**. This is for an **directed graph**.

Now that we know how to represent and store a **graph**, and have looked at the **complexity** of the fundamental operations we can perform on **graphs**, we are ready to start thinking about the kinds of problems we can solve using them. However, before we can really explore interesting applications of **graphs**, we first need to study a general **problem solving technique** that is very often used together with **graphs**, as an essential component of algorithms that use the **graph** to process information: **Recursion**.

5.4 Recursion as a tool for problem solving

The first thing to point out with regard to **recursion** is that you already know it, and have been using it for some time. Recall we made the point in the text above that both **linked-lists** and **BSTs** are **graphs**. And we just said that **recursion** is strongly tied to algorithms that work on **graphs**.

As we noted at the time, the operations we defined on **BSTs** in the previous section are recursive by nature. Consider the insertion of a new node into a **BST**: Start with the root node for the tree, check if it's NULL, and if not, decide whether the node should be inserted on the left or right subtree. Then recursively insert the node on the correct subtree. The process intuitively made sense from looking at the structure of the tree and thinking about how the insertion process had to work.

The same applies to **BST** search, deletion, and tree traversals. All of them are **recursive in nature**, which follows from the structure of the tree (remember we noted that every sub-tree of a **BST** is also a **BST**).

What we will do now is develop a general picture of what **recursion** is

- A way of thinking about problems that have particular structure
- A way to implement programs that solve such problems
- A tool for simplifying complex tasks that are difficult to handle otherwise

5.4.1 Examples of recursive problems and data structures

1) Graph search: **Graphs** are a **recursive** form of data representation. Every **sub-graph** of a **graph** is also a **graph** as illustrated in Fig. 5.12.

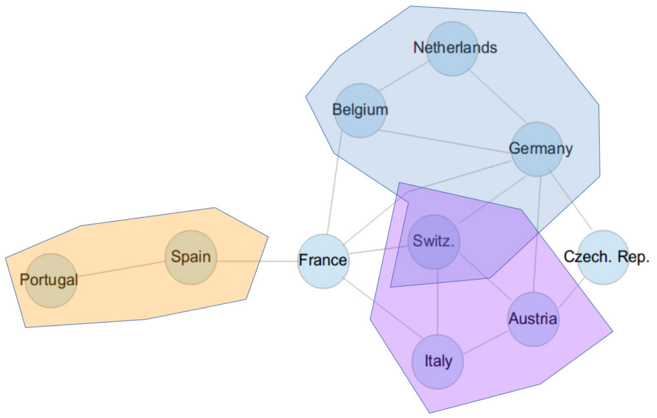


Figure 5.12: A **graph** and several possible **sub-graphs** (in colour), each **sub-graph** is itself a **graph** with its sets V of nodes and E of edges.

We had already seen this property with **BSTs**, but now we know it is a general property of **graphs**. **Linked-lists**, which are also **graphs**, are also recursive in nature: Every **sub-list** of a **linked-list** is also a **linked-list**.

And why is that interesting?

Because we can take advantage of the recursive structure of the **graph** to solve a seemingly complex problem by:

```
Taking the original input graph
 Breaking it up into smaller subgraphs
   Breaking those up into even smaller subgraphs
     And so on...
       Until the subgraphs are so tiny solving the problem is trivial
     Solve the smallest problem and use that solution to solve the slightly larger one
   And then the even larger one
 And then the even larger one
Until we have the solution to the original complex problem
```

This general process for looking for a specific **node** in a **graph** is an example of **recursion**. Variations of the process described above are used for **path finding**, **robot activity planning**, **scheduling**, image **pattern detection**, and many other applications. Let's see an example in **path planning**. The setup is as follows: any map whether it represents city streets, network routers available within some region, or in the case below, a little maze in a simulation, can be represented by a **graph** with **one node per location**, and **edges** linking together neighbouring locations that are connected (i.e. one can move from one such location to the next, there are no barriers in between).

In the case of city maps, the **nodes** can represent street intersections, and the **edges** can represent streets linking intersections together (**question:** would this be a **directed** graph or an **un-directed** graph?). For computer networks, the **nodes** would correspond to routers through which network traffic can flow, and the **edges** would correspond to data links between routers. In the example below, we have an 8×8 map, each location has been assigned a **node** in a **graph**, and the **nodes** are connected if the corresponding map locations are neighbours and there are no walls in between them. This is illustrated in Fig. 5.13.

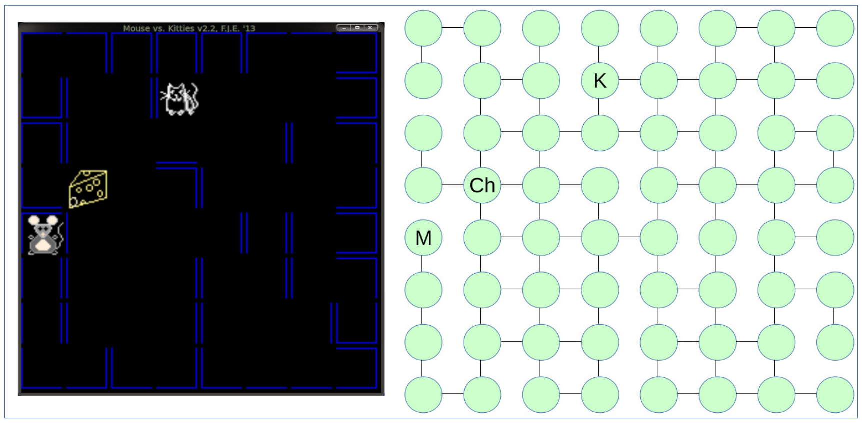


Figure 5.13: An example of a **graph** representing a small 8×8 maze. Each **node** corresponds to a location in the map, **edges** link together neighbouring locations that are connected (no walls between them), and represent the fact that someone walking around this maze could move from one location to another if the corresponding **nodes** in the graph are linked.

Problem: How do we find a path from the mouse to the cheese? In terms of our **graph** this means finding a path from the **node** labeled **M** to the **node** labeled **Ch**. For you, looking at the graph above, it's very easy to see the path the mouse should take. For a computer working on a **graph** this is not so simple.

✍ **Exercise 5.6** Develop **pseudo code** to solve this problem using only **iteration** (loops) but **no recursion**. You must follow the problem solving process we developed in Chapter 2, fully understand the problem, and develop a solution that is clear and detailed enough to serve as a basis for implementing a program to solve this task. Specifically, you should carefully consider how the **graph** will be processed (the order in which **nodes** will be considered, how to get information about neighbours, etc.) as well as **ow to keep track** of any information your process needs to form a complete path from the mouse to the cheese.

If you spend any time at all solving the previous exercise, you will quickly realize that using loops on **graphs** quickly leads to code that is (at best) long, cumbersome, full of special cases, and that will not work well on slightly different versions of the problem above. Since **graphs** are a **recursive ADT** by nature, you **should expect that non-recursive algorithms working on graphs will be cumbersome** and difficult to implement, test, and maintain. Conversely, the **recursive** solution is **simple**, **elegant**, and **easy to implement**.

The process is illustrated in Fig. 5.14 and we will take a detailed look at how it works later on in this Chapter.

2) Divide and conquer methods: Divide-and-conquer methods are behind some of the most powerful and useful algorithms we can find in computer science - including **binary search, quicksort**, the **Fast Fourier Transform** (used extensively in signal processing and signal analysis), **mergesort** (a sorting algorithm with a guaranteed **worst case complexity** of $O(NLog(N))$, and the **binary space-partitioning trees** for determining object visibility in computer graphics.

They are **naturally recursive** in that, by definition, they work by splitting a problem into smaller and smaller instances, applying to each of these the same algorithm, until the problem is easily solvable - then combining solutions for smaller instances of the problem to build the solution to larger and larger instances all the way back to the original one.

For example, mergesort is a sorting algorithm guaranteed to sort an input collection with complexity $O(NLog(N))$. The method works as follows (pseudo code):

```
mergesort(input_array)

   if the length of input_array is <= 1, then array is sorted: return input_array

   else
     split array into 2 sub-arrays: lower_half, and upper_half

     sorted_low = mergesort(lower_half)
     sorted_high = mergesort(upper_half)

     Merge in sorted order the two sub-arrays 'sorted_low' and 'sorted_high'
     to build the complete 'sorted_array'

     return sorted_array
```

We will look at this process in detail later on in this Chapter.

3) Computer graphics: Recursive structures are common in computer graphics, for example:

- Objects are often composed of parts (e.g. the various limbs of a person or animal, and different parts of plants or buildings). These are often modeled using tree-like structures - that brings us back to **graphs** which have recursive structure
- The rendering (drawing) process for certain plants like ferns and trees is naturally recursive because it involves drawing the same shape but at different sizes. This is shown in Fig. 5.15.

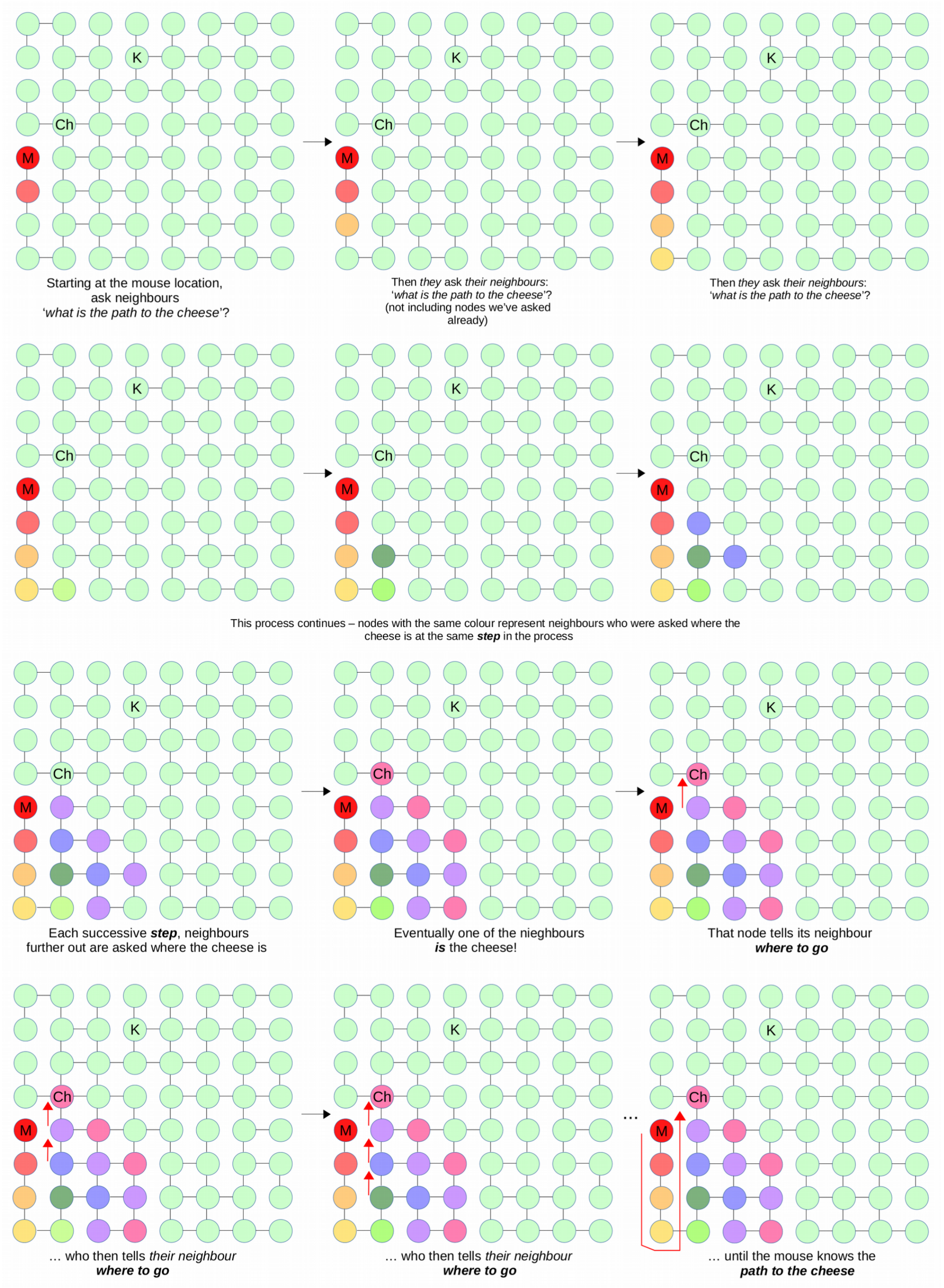

K
Ch
M
Starting at the mouse location, ask neighbours 'what is the path to the cheese'?
Then they ask their neighbours: 'what is the path to the cheese'? (not including nodes we've asked already)
Then they ask their neighbours: 'what is the path to the cheese'?
This process continues – nodes with the same colour represent neighbours who were asked where the cheese is at the same step in the process
Each successive step, neighbours further out are asked where the cheese is
Eventually one of the nieghbours is the cheese!
That node tells its neighbour where to go
... who then tells their neighbour where to go
... who then tells their neighbour where to go
...
... until the mouse knows the path to the cheese

Figure 5.14: Illustration of the **recursive** path-finding process for the maze example. At each step, nodes farther and farther away from the mouse are asked to find a path to the cheese. Eventually, one of the nodes is a neighbour of the cheese and knows where to go. The information then propagates backward neighbour to neighbour until the full path from the mouse to the cheese is obtained.

- Animation of objects routinely requires us to recursively apply transformations (changing the shape, size, orientation, or some other property of the object) to objects and their parts and sub parts - for example: To animate an arm, we move the upper arm, then the lower arm moves relative to the upper arm, then the hand moves relative to the lower-arm, and so on.
- The most advanced rendering methods (capable of creating movie-quality, photo-realistic scenes) are naturally recursive. They rely on tracing the path of light as it bounces from one object to another, then another, then another, and so on until the entire path of light from a light-emitting object to the camera has been found.

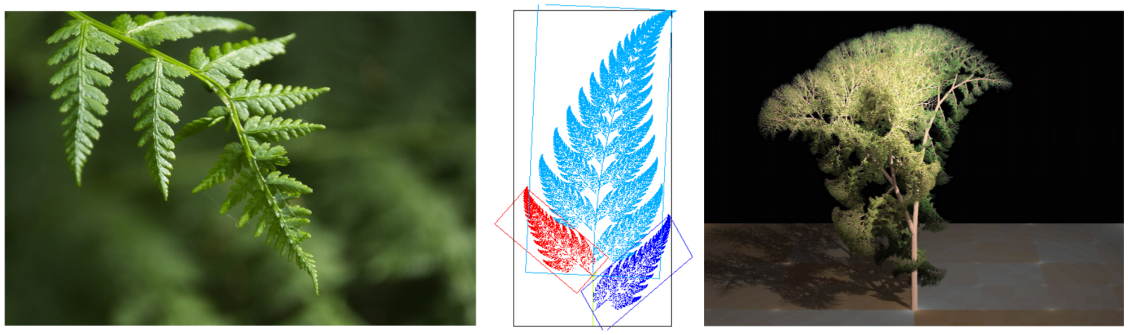


Figure 5.15: A real fern (left), a computer model of a fern (center) showing how the fern's parts are just smaller versions of the whole shape, and a computer rendering (right) created using a tree-based representation and a recursive rendering algorithm to create a life-like virtual plant. *Images: (left) Pixbay, used with permission, (center) A. M. Campos, Wikimedia Commons, public domain, (right) Solkoll, Wikimedia Commons, public domain.*

Recursion is a fundamental tool in computer graphics. It is used in some way for almost every component of the process of defining, representing, storing, and rendering computer generated images.

4) Programming languages: If you plan to be a serious software developer, you will need to learn different programming paradigms. **C** is based on the imperative programming model - program statements change the value of data and the state of the program in order to achieve the program's goal. Many other programming languages work in basically the same way, but change the syntax used as well as the features they offer to make your work as a developer easier.

The programs that **parse** and **compile** programs use **graph** representations and **graph methods** for representing the structure of instructions in the program, checking them for syntactical correctness, determining their meaning, and generating the relevant code. These tasks are often solved with **recursion**.

A different class of programming languages follow what is called a functional programming model (note that this doesn't have anything to do with using functions, the term has a different meaning here). Functional programming is built on the concept of a program being the result of the evaluation of a set of mathematical expressions or functions. Functional programming languages include LISP and its derivatives (Scheme, Racket), as well as Haskell. **Recursion** is often at the centre of such languages and is a fundamental part of how we have to think about problems if we want to approach them using functional programming.

5) File-system organization: As a final example of **recursive problems** and data structures, consider the organization of the file system in your computer. The information you have stored there is organized into folders, each of which will contain multiple items which themselves can be folders. The **directory structure** in the file system is in fact a tree (**graphs** again!), and is recursive. How would we go about finding a file, if we do not know in which folder it is stored? a general **recursive** solution would look like so:

```
// Find the name of the folder in which the requested file is stored

findFile(directory, file name)
  if a file matching 'file_name' is in 'directory'
    return 'directory'
  else
    for every 'sub_directory' in 'directory'
      directory_name = findFile(sub_directory, file_name)
      if 'directory_name' is not [empty]
         return 'directory_name'

      return [empty]
```

The above process travels through the directory structure in the computer, starting at an initially specified directory, looking for the file. For any given directory, if the file is not in that directory it then checks each of the sub directories recursively until either the file is found, or the entire directory structure in the computer has been searched.

The recursive structure of the file system is not limited to directories and files. For example, in Linux, the actual data blocks that make up a file are organized into a tree-structure in an **inode** (this has nothing to do with a certain company that sells phones, tablets, and computers) as shown in Fig. 5.16.

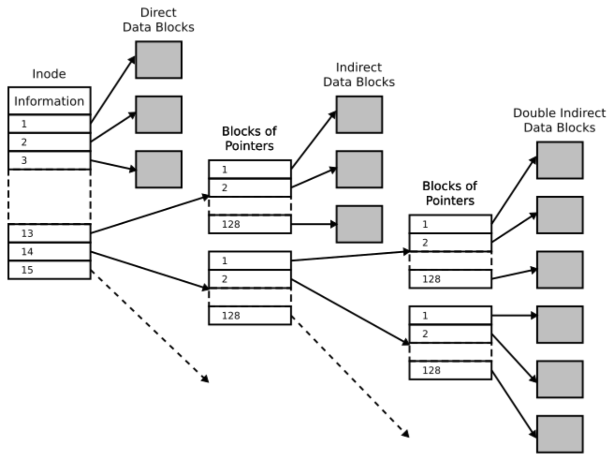


Figure 5.16: Structure of the data component of a Linux EXT2 File System **inode**. *Image: timtjtim, Wikimedia Commons, CC-SA4.0.*

5.5 General principles of recursion

By now you should have general idea of the kinds of problems that are suitable for solving with **recursion**, as a summary, we should consider using **recursion** to solve a problem when:

- The problem itself is complex - if there is a simple, direct solution using loops, you should go for that instead.
- The problem can be broken down into smaller or simpler versions of itself - e.g. a large **graph** can be broken up into smaller and smaller **subgraphs**.

- The nature of the problem is the same regardless of the size of the input - e.g. the problem of finding a path between two **nodes** is the same regardless of the size of the **graph**.
- At some point, the solution to one of the smaller problems becomes easy to compute - this allows us to stop breaking the problem down.
- We can use solutions to smaller problems to build the solution for larger problems easily.

This is fairly informal, so let us take a look at the components and process involved in working out a recursive solution to a problem.

5.5.1 Structure of a Recursive Solution

Because of the properties noted above, we can say that every recursive solution to a problem will have the following components:

1) The Base Case: This corresponds to the simplest (trivial) form of the problem we are solving, we should be able to easily and directly solve the problem for the base case without having to break the problem down any further. Every recursive solution **must have** at least one **base case** otherwise the recursive process keeps endlessly trying to simplify the problem and never returns a solution.

Example 5.1 Some examples of **base cases** for common applications of recursion include:

- For problems involving strings, the **base case** often is an empty string, or a string with one character - whatever the task, we can easily carry it out on a single character, and there is often nothing to do for the empty string.
- For numeric arrays, the **base case** often is an array with 1 entry, or an empty array - for instance, sorting an array with a single entry is trivial.
- For graph problems, the **base case** often involves a **graph** with a single node, or a graph where the node being processed has a specific property - for instance, for mapping applications, it could be that the node being processed represents the location we are looking for.
- For information search (on lists, trees, or other data structures) the **base case** can be an empty data structure, or having found the node with the information we need.

Note

As you can see there can be several different **base cases** for a given problem. It is important to **consider all the possible base cases** that apply. In the examples above we have listed a few of the common **base cases** for a few common problems. But notice that we only say these apply **often**. For a particular problem, they may not apply, or there may be additional **base cases** not listed above. The importance of making sure all the relevant **base cases** have been accounted for is that, when one or more of them are missing, there would be inputs for which our recursive solution would not work. It would reach one of the missing **base cases** and just keep going endlessly (which in practice often results in the program crashing).

2) The Recursive Case: This corresponds to the general form of our problem, which we have to split, simplify, or otherwise make smaller in order to get one of the **base cases**. The interesting part of solving a problem with **recursion** involves thinking about how to break down our specific problem. After we have split the problem in an appropriate way into two or more sub problems, we then **recursively solve** the smaller sub problems. The **recursive case** implementation is also responsible for **building a solution** for the original problem from the solutions to the separate sub problems we split it into.

Example 5.2 Here are a few examples of how different types of problems can be broken down

- For strings, the **recursive case** often consists of breaking the string into chunks. Some problems will require taking out a particular character (e.g. the first, or the last one), others will split the string in chunks at a specific point. Either way, if we keep splitting each chunk we will eventually reach the **base case**.
- For numeric arrays, the **recursive case** often splits the array into a pair of sub-arrays. Some problems split the first or last entry from the rest of the array, others (like mergesort) split the array in two. Either way the resulting sub-arrays are closer to the base case.
- For graphs, the **recursive case** will often perform processing on subgraphs - for example, in the path-finding problem discussed above, the process considers at each step only a small set of **neighbours** that have not yet been checked for a path to the destination. Another possibility is that the **recursive case** may split the graph into disjoint subsets and process each of them recursively (similarly to how mergesort processes an array). Examples of this would be **tree traversals**. The traversal process splits a tree into left and right subtrees, and processes these recursively until reaching the base case of an empty subtree.
- For search, the **recursive case** often involves searching over a subset of the data structure. For example, for **BSTs**, the search process determines which sub-tree the desired item should be in, and recursively searches that subtree. At each step, the remaining sub-tree is closer to the base case (either finding the item we want, or reaching an empty subtree).

Note

It is essential for the **recursive case** to bring the problem closer to a **base case**. If the problem is not getting easier to solve despite repeated application of the **recursive case**, or if it takes too many steps to reach a **base case**, we should carefully re-think our process. As we shall see in short order, the choice of how to break down a problem for the **recursive case** can have a significant impact on the **computational complexity** of our **recursive** solution.

5.5.2 How to design a recursive solution

The first step of the process is exactly as discussed at the end of Chapter 2 - thoroughly understand what the problem is, write down and illustrate an example - e.g. if you are working on arrays, draw a representative sample array with data, and consider if the problem has the requisite properties that make it a good candidate for a **recursive** solution.

Then develop the algorithm **in pseudo code** as per the discussion in Chapter 2, paying special attention to:

- What are the the **base cases** for this problem? It is important to work out **on paper** a few examples of how small versions of the problem may look, and ensure that no **base cases** are missing.

- Once you we have the **base cases**, determine how you could split an instance of the problem that is **not a base case** into **smaller or simpler sub problems** that will get closer to the base case - this gives you the **recursive case**. At this point it is also important to work out how to build the solution for the original problem from the solutions to smaller sub problems. Once more **work out a few examples on paper** and make sure the process works, this will often allow you to find any missing **base cases**.
- Consider different ways of splitting the problem for the **recursive case**. Try to figure out if different choices make your solution do more or less work in terms of the number of times the **recursive case** needs to be applied, and in terms of the **complexity** of building bigger solutions from those for smaller sub problems.
- Your pseudo code algorithm should clearly involve **recursively solving sub problems**.

Note

Recursive solutions that have not been carefully constructed can be difficult to **test**, **trace**, **debug**, and **maintain**. So special care has to be taken to ensure the proposed algorithm is **clean - with reasonable and easy to understand base and recursive cases**, **carefully thought out**, and **well documented** - this means carefully noting design choices and your rationale for how the **recursive case** was selected.

Let us see how the process described above can be applied to solve a very common problem: **sorting** a large collection of information.

5.5.3 Sorting data recursively

Let us consider the problem of **sorting data stored in an array**. In the discussion below the array contains **integers** but this is not a limitation, any data for which we can implement a meaningful **comparison function** can be sorted in the same way. Our goal will be to develop a **recursive** solution by following the process described above, and to study the effect of different ways to implement the **recursive case** on the **computational complexity** of the resulting sorting algorithm.

The Problem: Given an input array of **un-sorted** data, return an array with the same number of entries **sorted** in ascending order. We will work with the following sample array:

```
    [ 9 | 6 | 3 | 7 | 0 |2 | 8 | 1 | 4 | 5 ]
```

this is a very small array, but it is enough for us to understand the problem and develop our **recursive** solution.

Step 1) Find the base cases: We can figure out what these should be by considering the easiest instance of our problem. For **sorting**, it doesn't get any easier than having an **empty array** - there is nothing to do, since it contains no data. It is a valid **base case** and we will see shortly that it comes up during the process of sorting non-empty arrays. There is another **base case** corresponding to an array with a single entry. This is a base case because a single-entry array is **already sorted**.

Any array that has more than one entry may need to be sorted, it can also be split into smaller arrays that are closer to one of the two **base cases** above. Therefore **arrays with more than 1 entry are not base cases**.

2) The recursive case: As noted above, there are many ways in which we could split our problem into sub problems that are closer to the base case. We also noted that depending on the choice we will end up with simpler/more intuitive solutions, or with more complex solutions. And that the different possibilities may result in algorithms that are less or more efficient.

Let's have a look at three different ways we could split the problem of sorting an array, each of which leads to a different sorting method. We will study the complexity of the sorting process for each of the splitting methods, and realize that seemingly small differences in how the **recursive case** works can have a large effect on **complexity**.

```
 Algorithm: sort_case1()

 // Recursively sort an array of integers

 Input: An 'array' of length N containing integers
 Output: A 'sorted' array of integers of length N

If 'array' is a base case: {empty array, array of size 1}

      return 'array'    // Base case - it's already sorted!

Otherwise

      Split 'array' into 'sub1' and 'sub2' such that:

          // Recursive case #1: split the array into first entry and everything else

          sub1=array[1]             // First entry in the array
          sub2=array[2:end]         // All remaining entries in the array

      Recursively sort 'sub1' and 'sub2' using recursive case #1

          subsort1=sort_case1(sub1)
          subsort2=sort_case1(sub2)

      Merge the sorted sub-arrays in order to build the complete
      sorted array corresponding to the input

          sort=merge_in_order(subsort1, subsort2)
          return 'sort'
```

Question: Does the **recursive** case work? does it eventually reduce an array of any size to sub-arrays that are one of our two **base cases**?. The splitting process will be applied to any array of size 2 or greater. It will split the array into two **smaller** sub-arrays. These will undergo further splitting as needed. At each step the resulting sub-arrays are smaller, so eventually we must arrive at one of our **base cases**.

Let's see the process in action, on the small input array shown above:

```
    [ 9 | 6 | 3 | 7 | 0 | 2 | 8 | 1 | 4 | 5 ]                // Original array

    [9] : [6 | 3 | 7 |0 | 2 | 8 | 1 | 4 | 5 ]                // First split

    // [9] is in base case, but we have to wait for the second
    // sub-array to be sorted in order to re-build the complete
    // sorted array

    [9] : [6] : [3 | 7 | 0 | 2 | 8 | 1 | 4 | 5 ]             // Second split
```

```
// [6] is in base case, but we need to wait for the second
// sub-array to be sorted... [9] is still waiting!

[9] : [6] : [3] : [7 | 0 | 2 | 8 | 1 | 4 | 5 ]             // Third split

// [3] is in base case, need to wait for the second sub-array
// to be sorted. [9] and [6] are still waiting for a result

.
.      // splitting continues...
.

[9] : [6] : [3] : [7] : [0] : [2] : [8] : [1] : [4 | 5]    // Eight split

// 9, 6, 3, 7, 0, 2, 8, and 1 are waiting, need to sort
// [4 | 5] which is not in base case

[9] : [6] : [3] : [7] : [0] : [2] : [8] : [1] : [4] : [5]  // Final split

// [4] and [5] are base case, so the recursive calls simply
// return the same two sub-arrays and they can be merged
// (they are merged in sorted order)

[9] : [6] : [3] : [7] : [0] : [2] : [8] : [1] : [4 | 5]    // First merge

// [1] was waiting for [4 | 5] to be sorted, it is now
// sorted so it gets merged with [1]

[9] : [6] : [3] : [7] : [0] : [2] : [8] : [1 | 4 | 5]      // Second merge

// [8] was waiting for [1 | 4 | 5] to be sorted, it is
// now sorted so it gets merged with [8]

[9] : [6] : [3] : [7] : [0] : [2] : [1 | 4 | 5 | 8]        // Third merge

.
.      // merging continues...
.

[9] : [6] : [0 | 1 | 2 | 3 | 4 | 5 | 7 | 8]                // Seventh merge

// [6] was waiting, it can now be merged into a larger
// sorted array

[9] : [0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8]                  // Eight merge

// Finally, [9] was waiting, and it can be marged with the
// remaining (now sorted!) entries to produce the final
// sorted array

[0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9]                    // Final merge
```

As you can see, the process itself is fairly straightforward. The array is reduced to individual entries one element at a time, until all the sub-arrays are in **base case**, then they are merged in order one by one to build the final sorted array.

Note

Merging two sorted sub-arrays with **N** entries in total into a single sorted array requires $O(N)$ time. The process is as follows:

```
// Procedure merge_in_sorted_order()
// input: two sorted sub-arrays 'sub1[]' and 'sub2[]'
// output: a single 'sorted[]' array containing the entries from both sub-arrays

out_index=0;       // Indexes into the output array and each
sub1_index=0;      // input sub-array. Initially 0 (first entry)
sub2_index=0;

while out_index is less than N      // This loops exactly N times

    choose whichever entry is smaller between

        sub1[sub1_index] and sub2[sub2_index]

    copy the chosen entry to sorted[out_index]

    if the entry from sub1 was selected, increment sub1_index by 1
     otherwise, increment sub2_index by 1

    increment out_index by one
```

Complexity of sorting using recursive case #1: Let's see how much work needs to be done by the sorting process when we split arrays according to **recursive case #1**.

- The **recursive case** has to perform **N-1** splits and **N-1** merges - this is because each split except for the last one leaves a single entry of the array in the **base case** (the last split takes care of two entries). So there is a factor of **N** here.
- For an array of size **N**, the split has a cost of $O(N)$ because **N** elements have to be placed in sub-arrays (copying the **N** entries into sub arrays requires **N** copy operations).
- Combining two **sorted** sub-arrays with **N** entries (combined) into a single **sorted** array of size **N** has a cost of $O(N)$.
- Therefore, each split together with the corresponding merge has a cost of $O(N)$.
- Since there are **N-1** split/merge steps, each with a cost of $O(N)$, the total cost for the sorting process is $O(N^2)$.

This means that the recursive sorting algorithm based on **recursive case #1** is just as slow as **bubble-sort**. This is not a good result. Looking at the example above with the 10-entry array, we note that it takes **a lot of splitting and merging** to get the job done. This is because **recursive case #1** reduces the size of the problem by **1** at each step. What can we do to reduce the number of steps required to reduce the sub-arrays to a **base case**?

```
Algorithm: sort_case2()
```

```
 // Recursively sort an array of integers

 Input: An 'array' of length N containing integers
 Output: A 'sorted' array of integers of length N

If 'array' is a base case: {empty array, array of size 1}

      return 'array'    // Base case - it's already sorted!

Otherwise

      Split 'array' into 'sub1' and 'sub2' such that:

         // Recursive case #2: split the array in two halves

         m=floor(N/2)               // Half the size of the input array, rounded down.
         sub1=array[1:m]            // First half of the array
         sub2=array[m+1:end]        // Second half of the array

      Recursively sort 'sub1' and 'sub2' using recursive case #1

         subsort1=sort_case2(sub1)
         subsort2=sort_case2(sub2)

      Merge the sorted sub-arrays in order to build the complete
      sorted array corresponding to the input

         sort=merge_in_order(subsort1, subsort2)
         return 'sort'
```

Let's see how the above process works on the same input array we used with **recursive case #1**:

```
  [ 9 | 6 | 3 | 7 | 0 | 2 | 8 | 1 | 4 | 5 ]                // Original array

                     (A)
  [9 | 6 | 3 | 7 | 0] : [2 | 8 | 1 | 4 | 5]                // First split

  // Both arrays are not base case, each has to be sorted using
  // recursive case #2, this means 9,6,3,7,0 will be split in
  // half, and 2,8,1,4,5 will be split in half (splits are
  // labeled B for clarity)

        (B1)                   (B2)
  [9 | 6] : [3 | 7 | 0] : [2 | 8] : [1 | 4 | 5]            // Second split

  // All the sub-arrays are still not base case, so each will be
  // sorted using recursive case #2 and split in half again
  // (splits are labeled C for clarity)

     (C1)        (C2)             (C3)        (C4)
  [9] : [6] : [3] : [7 | 0] : [2] : [8] : [1] : [4 | 5]     // Third split

  // The splits C2 and C4 still have one sub-array (each) that
  // is not a base case, so one more round of recursive case #2
  // is applied (with splits labeled D for clarity)

                       (D1)                          (D2)
  [6] : [9] : [3] : [7] : [0] : [2] : [8] : [1] : [4] : [5] // Final split
```

```
// Reached base case for all sub-arrays, now we just merge
// in the reverse order - first the D splits merge

[6]: [9] : [3] : [0 | 7] : [2] : [8] : [1] : [4 | 5]      // First merge

// Now the C splits merge

[6 | 9] : [0 | 3 | 7] : [2 | 8] : [1 | 4 | 5]             // Second merge

// Now the B splits merge

[0 | 3 | 6 | 7 | 9] : [1 | 2 | 4 | 5 | 8]                 // Third split

// And finally the A split merges to create the final ourput array

[0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9]
```

You can probably already see that **recursive case #2** takes fewer steps to reach the **base case** for all the sub-arrays. Since at each step, the size of the arrays is halved, the number of steps required to reach the **base case** is $Log_2(N)$ - the progressive halving of the size of the input is the same process that allows **binary search** to achieve $O(Log(N))$ search complexity and that allows **BSTs** to (on average) provide $O(Log(N))$ complexity for search, insert, and delete.

Complexity of sorting using recursive case #2: Let's see how much work needs to be done by the sorting process when we split arrays according to **recursive case #2**.

- The **recursive case** has to perform $Log_2(N)$ splits and $Log_2(N)$ merges - this is because each split reduces the size of the input by a factor of 2, and each merge produces a sorted array twice the size of the input sub-arrays.
- For an array of size **N**, the split has a cost of $O(N)$ because **N** elements have to be placed in sub-arrays (copying the **N** entries into sub arrays requires **N** copy operations).
- Merging two **sorted** sub-arrays with **N** entries (combined) into a single **sorted** array of size **N** has a cost of $O(N)$.
- Therefore, each split together with the corresponding merge has a cost of $O(N)$.
- Since there are $O(Log_2(N))$ split/merge steps, each with a cost of $O(N)$, the total cost for the sorting process is $O(N \cdot Log(N))$.

This is a great result - the sorting algorithm that uses **recursive case #2** has a guaranteed **worst case complexity** of $O(N \cdot Log(N))$. You will recognize that this is in fact **merge-sort**, a solid, reliable, and easy to implement method that shows the power of **recursion**. It should give you a moment of pause to consider that the two sorting algorithms we discussed thus far are **identical** save for the **choice of recursive case**. Splitting the array in two rather than one-versus-the-rest gives us an algorithm whose complexity matches the best known complexity for general sorting algorithms.

There is one more choice of **recursive case** that is worth thinking about:

```
 Algorithm: sort_case3()

 // Recursively sort an array of integers

 Input: An 'array' of length N containing integers
 Output: A 'sorted' array of integers of length N

If 'array' is a base case: {empty array, array of size 1}

      return 'array'    // Base case - it's already sorted!

Otherwise

      Split 'array' into 'sub1' 'pivot' and 'sub2' such that:

          // Recursive case #3: split with regard to a pivot entry

          choose m=random entry in 0:N-1 // Select a random entry
          pivot=input[m]                 // The pivot is the value
                                         // of this entry
          sub1 := All entries in input whose value < pivot
          sub2 := All entries in input whose value > pivot

      Recursively sort 'sub1' and 'sub2' using recursive case #3

          subsort1=sort_case3(sub1)
          subsort2=sort_case3(sub2)

      Merge the sorted sub-arrays in order to build the complete
      sorted array corresponding to the input

          sort=merge_in_order(subsort1, pivot, subsort2)
          return 'sort'
```

This process doesn't split the input into sub-arrays of a pre-defined size. Instead, it chooses a random entry in the input array (called the **pivot**) and then splits the input into two sub-arrays consisting of the elements that are **lesser than the pivot** and the entries with value **greater than the pivot**, plus the **pivot** itself (notice this is a three-way split). The splitting process **does not sort the entries in each sub array** - but it does **sort the input values with regard to the pivot**.

Let's see how **recursive case #3** works on the sample input array:

```
    [ 9 | 6 | 3 | 7 | 0 | 2 | 8 | 1 | 4 | 5 ]                    // Original array

        m=2, pivot=3 (input[2]=3)
                     (A)
    [ 0 | 2 | 1] : [3] : [9 | 6 | 7 | 8 | 4 | 5]                  // 1st split

    // Notice the sub-arrays are not the same size, and that
    // within each sub-array the entries are not sorted, but
    // each set is sorted w.r.t. the pivot. The pivot is
    // now in the correct sorted spot!
    // The two sub-arrays are not base-case, we have to
    // apply recursive case #3 to each of them (the splits
    // are labeled B for clarity)

        m=1, pivot=2           m=5, pivot=5                        // 2nd split
```

```
             (B1)                    (B2)
[0 | 1] : [2] : [] : [3] : [4] : [5] : [9 | 6 | 7 | 8]

// Each split will have a different pivot. We still have
// several arrays not in base case, so apply
// recursive case #3 again (splits labeled C for
// clarity - pivot is under the label)

    (C1)                                                (C2)
[] : [0] : [1] : [2] : [] : [3] : [4] : [5] : [6 | 7] : [8] : [9] // 3rd split

// Need one more round of recursive case #3 to bring
// everything to base case (splits labeled D for
// clarity, pivot is under the label)
                                                    (D1)
[] : [0] : [1] : [2] : [] : [3] : [4] : [5] : [6] : [7] : [] : [8] : [9] // Done!

// At this point we are done splitiing, merge sub-arrays
// in reverse order. Start with D splits

[] : [0] : [1] : [2] : [] : [3] : [4] : [5] : [6 | 7] : [8] : [9] // 1st merge

// Then the C splits

[0 | 1] : [2] : [] : [3] : [4] : [5] : [6 | 7 | 8 | 9]            // 2nd merge

// Then B splits

[0 | 1 | 2] : [3] : [4 | 5 | 6 | 7 | 8 | 9]                       // 3rd merge

// And finally the A split to yield the complete sorted result

[0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9]                          // Final merge
```

This sorting method is called **quicksort** and is one of the most commonly used, and most reliable general sorting algorithms. As the name implies, it is often **fast**, but it's not clear just from the example above why that would be the case.

However, you may be able to see a few features of **quicksort** in the example above: The splits are not symmetric - they can happen anywhere along the array. Sometimes they are close to the **one-versus-the-rest** case, sometimes they are closer to the **split-in-half** case, and sometimes we even have an empty sub-array on one side or the other from the **pivot**. It all depends on luck since we choose the **pivot** randomly for each split. You can also see that **recursive case #3** appears to do less splitting and merging than **recursive case #1**, but it's not clear just how much more or how much less work it does compared to **merge sort**.

So the question is: What can we say about the complexity of sorting with **quicksort**?

In the **worst-case**, we get very unlucky with **each and every split** and choose the **pivot** that corresponds either to the smallest, or the largest value in the array. The result is one empty sub-array, the 1-entry array with the **pivot** itself, and one sub-array that contains the rest of the entries in the original. This is identical to **recursive case #1**, and results in an algorithm that performs **N-1** splits and **N-1** merges, with a computational cost of $O(N^2)$. The **worst case complexity** of sorting an array using **quicksort** is $O(N^2)$.

That doesn't seem good - why is **quicksort** so popular, so often used, and said to be **fast** when we have just seen that it's **worst case complexity** is just as bad as **bubble-sort**?

The usefulness of **quicksort** results from what happens **in the average case**. The question we need to ask is **what is the probability that we hit the worst-case split**? For an array of size **N**, choosing the smallest or largest value by random chance has a probability of $2/N$. For a large **N** this is very very small! - of course, we could worry about splits that are close to the worst case, even if not the very worst, but it turns out even if we pick the smallest 100, and the biggest 100 pivot values, the probability of randomly selecting one of them is $200/N$ which again, for a large value of **N** will be very small.

Not only that, but the **worst case complexity** of **quicksort** happens when we consistently get the worst split - at every step of the sorting process. The probability of that happening with **random pivot selection** is astronomically small for a large input array.

So in practice, we don't have to worry much about encountering the **worst case** for **quicksort**. But that doesn't tell us anything about what to expect **on average**. The mathematical analysis of **quicksort** is beyond the scope of this Chapter, however, we can gain an understanding of why **quicksort** works so well by studying the results of simulating **quicksort** millions of times (with different, randomly generated, unsorted input arrays) for different values of **N**. Fig. 5.17 shows the results of simulating the **quicksort** splitting process $1,000,000$ times for arrays of size $N = 10,000$ and **counting the number of splits required to reach base case for all sub-arrays**.

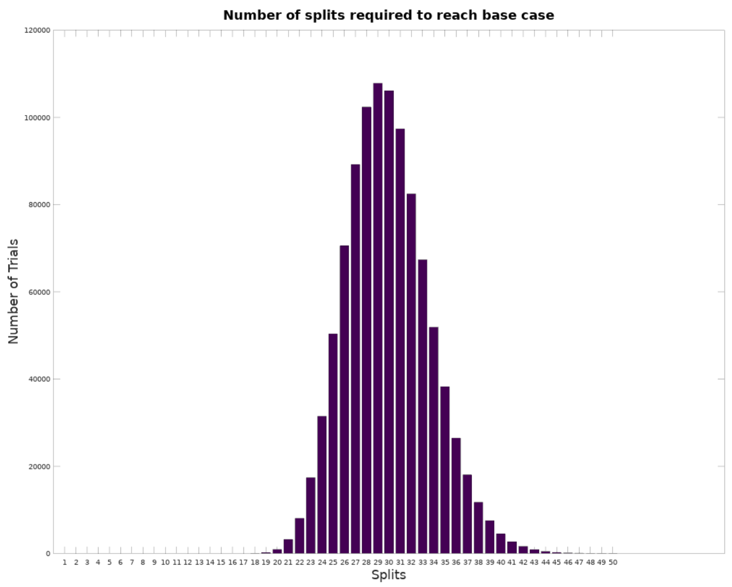


Figure 5.17: Number of levels required to reduce arrays of size $N = 10,000$ to **base case**. One million trials with different random arrays were simulated for this graph.

What we can see from Fig. 5.17 is that while the number of splits is **not** $Log_2(N)$ which for $N = 10,000$ is just over 13 splits, the maximum number of splits that were required over the entire set of one million different arrays is still pretty small (close to 50 which is tiny compared to the value of **N**). This indicates that **quicksort can reach the base case for all sub-arrays quickly**, without too many splits.

An even more interesting and encouraging picture emerges once we simulate the **quicksort** splitting process for different values of **N**, and plot the **average number of splits** required to bring all the sub-arrays to **base case**. This is shown in Fig. 5.18.

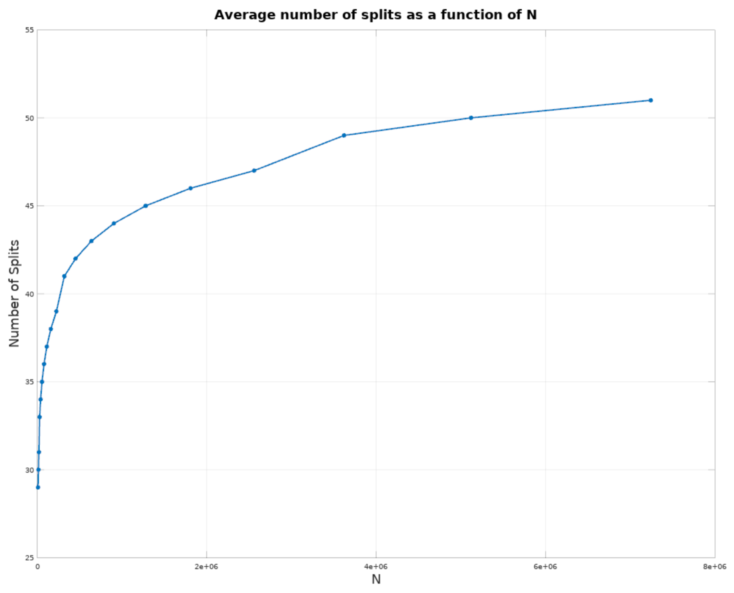


Figure 5.18: Average number of splits required to reduce arrays of size N to **base case** for N from $10,000$ to just under $8,000,000$. One million trials with different random arrays were simulated for each value of N in the graph.

The shape of the graph in Fig. 5.18 should be familiar - it is a **logarithmic** curve, and illustrates a well known result: The **average case complexity** of **quicksort** is $O(N \cdot Log(N))$. This is because **on average**, **quicksort** will carry out a number of splits and merges proportional to $Log(N)$, and each of these has a cost of $O(N)$. This is a very good result. In the **average case**, **quicksort** has the same complexity as **mergesort** and achieves the best known complexity for general sorting methods.

Note

You may be concerned that **quicksort** does more work during the splitting step than **mergesort** - it needs to compare each entry in the input array against the pivot and decide which sub-array it should go into. Conversely, **mergesort** just splits the arrays in half without additional work.
However, the extra work **quicksort** does when splitting evens out during the merging step - you can see in the example above that merging sub-arrays sorted with **quicksort** is easy because all the entries are in the correct location, **no additional comparisons are needed**. Conversely, **mergesort** has to compare entries from the sorted sub-arrays during merging in order to build the larger sorted array.

The conclusion from the above is that **quicksort** is **competitive** with **mergesort** and other **efficient sorting methods**. Both **quicksort** and **mergesort** are readily available in all common programming languages via standard libraries, you can use them for any sorting tasks you may encounter while solving interesting problems, and now you understand how they work. Both of them are **recursive**, and the central difference between the algorithms is **how the recursive case is handled**. We have seen that the choice of **recursive case** matters greatly, and can mean the difference between an **efficient** algorithm, and a **slow, impractical** one.

We have also gone through the process of designing a **recursive solution** for a general problem: **Sorting**. This should help you when you find a need to approach other problems with **recursion**. Next, we should spend some time considering the performance implications of **recursive algorithms**, as they deserve careful thought and should not be ignored when considering whether or not **recursion** is the right tool for a specific task or problem.

5.6 Implementation considerations for recursive methods

Recursion requires a function to call itself a (possibly large) number of times. Recall, from the memory model in Chapter 2, that every time call a function is called, **space has to be reserved for** the function's **parameters, variables, and return value**. In non-recursive code, we don't usually worry about it because we know that after a function is called, its work is completed, and it returns a value, the space reserved for the function is released.

However, with recursive code this can become a problem unless we are careful - at any given time, there will be multiple (and possibly many) **active calls** to the recursive function. These are function calls that are **waiting from a result** from another recursive call, and until they receive that result, they hang around in memory and occupy space.

To understand this problem, let's have a look at a short example of a function that sums up an array of integers recursively (as we've noted before, this is not a problem where we would immediately think of recursion, but it's simple enough it will help us illustrate what happens in memory with recursive function calls).

The recursive function that computes the sum of the entries in an array works as follows (pseudo code):

```
sum(array, n)                                    : L1
  if (n==0) return array[0];                     : L2
  else return array[n] + sum(array, n-1)         : L3
```

The function takes an input array, and recursively computes the sum of its elements. The **base case** is an array with a single entry in which case the sum is simply the value for that element.

The input parameter **n** is used to indicate the element of the array that the function is adding at a specific point in the process - it is intended to start at the end of the array and work backward toward the first element in the array. The **recursive case** simply decreases **n** by one, which has the effect of reducing the size of the array at each step. Once the **base case** is reached, the recursion rebuilds the sum from the first element back to the last.

The recursive process for a small input array would look as follows:

```
  sum([10,5,3,2,1,8],5)                     // First call, n=5, not base case       (A)
      sum([10,5,3,2,1,8],4)                 // recursive call with n=4, not base case (B)
          sum([10,5,3,2,1,8],3)             // recursive call with n=3, not base case (C)
              sum([10,5,3,2,1,8),2)         // recursive call with n=2, not base case (D)
                  sum([10,5,3,2,1,8],1)     // recursive call with n=1, not base case (E)
                      sum([10,5,3,2,1,8],0) // recursive call with n=0, base case!  (F)

                      // At this point there are 6 active calls to 'sum()', each with
                      // their own input parameters, local variables, and space for
                      // a return value! - these correspond to (A) - (F)

                      // The last call can be completed, so (F) returns 10 to (E)
                      // space for (F) is released

                  (E) - can now be completed, returns 5+10, space for (E) is released
              (D) - can now be completed, returns 3+15, space for (D) is released
          (C) - can now be completed, returns 2+18, space for (C) is released
      (B) - can now be completed, returns 1+20, space for (B) is released
  (A) - finally, the original call can be completed, returns 29, and space for (A) is
      released
```

The same process is illustrated in Fig. 5.19 where you can see how memory is reserved for each function tall inside a **call stack** - this is a region of the computer memory that is set aside for the function calls. Any memory requested by any of the functions in the program is reserved in the **stack**, and it is reserved directly **on top** of any memory being used by other **active calls**. The specific region of memory assigned to each function call is referred to as the function's **stack frame**. In Fig. 5.19, each block (shown as a rectangle) for one function call represents a separate **stack frame**.

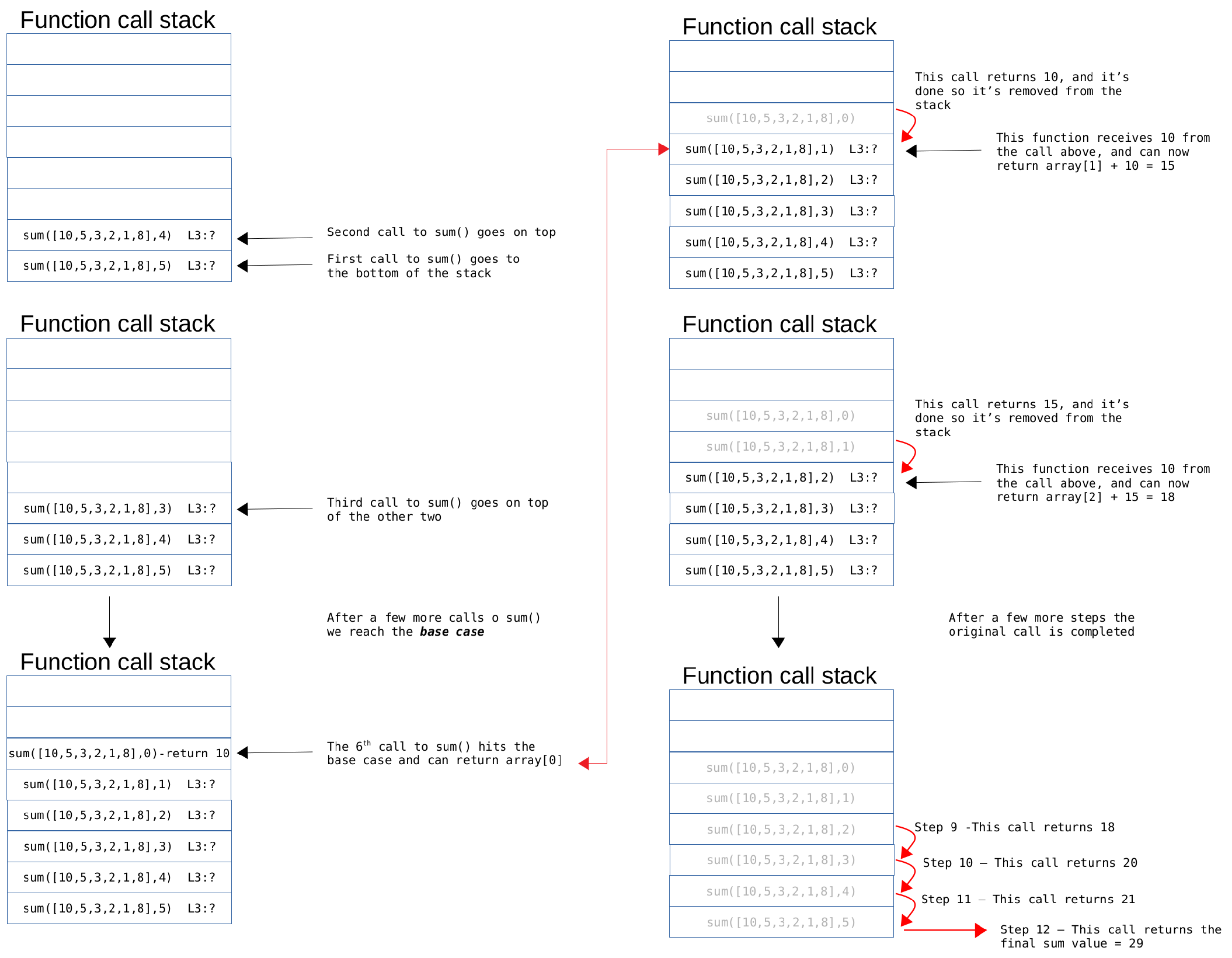

Function call stack
sum([10,5,3,2,1,8],4) L3:?
sum([10,5,3,2,1,8],5) L3:?
Second call to sum() goes on top
First call to sum() goes to the bottom of the stack
Function call stack
sum([10,5,3,2,1,8],3) L3:?
sum([10,5,3,2,1,8],4) L3:?
sum([10,5,3,2,1,8],5) L3:?
Third call to sum() goes on top of the other two
After a few more calls o sum() we reach the ***base case***
Function call stack
sum([10,5,3,2,1,8],0)-return 10
sum([10,5,3,2,1,8],1) L3:?
sum([10,5,3,2,1,8],2) L3:?
sum([10,5,3,2,1,8],3) L3:?
sum([10,5,3,2,1,8],4) L3:?
sum([10,5,3,2,1,8],5) L3:?
The 6th call to sum() hits the base case and can return array[0]
Function call stack
sum([10,5,3,2,1,8],0)
sum([10,5,3,2,1,8],1) L3:?
sum([10,5,3,2,1,8],2) L3:?
sum([10,5,3,2,1,8],3) L3:?
sum([10,5,3,2,1,8],4) L3:?
sum([10,5,3,2,1,8],5) L3:?
This call returns 10, and it's done so it's removed from the stack
This function receives 10 from the call above, and can now return array[1] + 10 = 15
Function call stack
sum([10,5,3,2,1,8],0)
sum([10,5,3,2,1,8],1)
sum([10,5,3,2,1,8],2) L3:?
sum([10,5,3,2,1,8],3) L3:?
sum([10,5,3,2,1,8],4) L3:?
sum([10,5,3,2,1,8],5) L3:?
This call returns 15, and it's done so it's removed from the stack
This function receives 10 from the call above, and can now return array[2] + 15 = 18
After a few more steps the original call is completed
Function call stack
sum([10,5,3,2,1,8],0)
sum([10,5,3,2,1,8],1)
sum([10,5,3,2,1,8],2)
sum([10,5,3,2,1,8],3)
sum([10,5,3,2,1,8],4)
sum([10,5,3,2,1,8],5)
Step 9 -This call returns 18
Step 10 – This call returns 20
Step 11 – This call returns 21
Step 12 – This call returns the final sum value = 29

Figure 5.19: Illustration of how memory space is reserved in the **function call stack** for the recursive function calls to **sum()**. Each successive call goes to the **top of the stack**, previous calls remain **active** - they are waiting for a result and can not finish their work until the recursion returns the values they need.

There are two important implications from the way the process shown above develops:

- Recursive function calls take up space. Depending on the **depth of the recursion**, which is the number of calls required until the **base case** is reached, the space required may be significant. For an array of size N, the **sum()** function above will require N separate chunks of space in the **call stack**.
- Reserving space in the **function call stack** and releasing it when function calls return requires **work**. It is done automatically, but the computer still takes some small amount of time to reserve space and then to

release it when no longer needed. Once again, this may become significant if the **recursion depth** is large.

These two factors have important implications for **recursive functions**: They require significantly more memory than **non-recursive** functions, and in fact if we are not careful we run the risk of running **out of memory in the call stack** which causes a program to crash. Secondly, they are **slower** than **non-recursive** functions because of the extra work required to reserve and later release memory from the **call stack**.

When used for problems that have a **recursive nature** (as explained in detail in Section 5.5), the additional memory requirements and small overhead of handling memory in the **call stack** are balanced by the simplicity, elegance, and intuitive nature of the **recursive solution** - in such cases, a **non-recursive** solution is likely to be much more involved, harder to understand, maintain, test, and debug; and likely will require its own additional memory in order to keep track of information that the **recursive** solution automatically maintains via the **call stack**.

However, for certain problems that could equally well be solved with and without **recursion**, the choice of whether or not to implement a **recursive** algorithm should have carefully considered the extra space and slower nature of the recursive process.

Note

It is important to remember that for every problem we can solve with a computer, it is possible to find a **recursive** solution, as well as a **non-recursive** solution. Though for many cases it may not be obvious how to develop one or the other. There is no in-principle limitation to what kind of problems can be solved **recursively**, and the same is true for **iterative** (non-recursive) methods.

5.6.1 Tail-recursion optimization

The extra memory required for **recursion**, and the overhead of managing the **function call stack** can sometimes be avoided by using a technique called **tail-recursion optimization**.

In the example above, the **sum()** function builds the result we want from **the partial sums returned by recursive calls** - the last recursive call (the one that reaches the base case) returns a value, which then is used to compute and return the next partial sum, which then is used to compute and return the next partial sum, and so on all the way back to the first call to **sum()** which computes and returns the final result.

This line of pseudo code takes care of this part:

```
else return array[n] + sum(array, n-1)    : L3
```

The thing to notice is that once the recursive call to **sum()** returns a value, we **still need to add array[n]** before we can return a result. Because the function still has work to do after the recursive call is complete, we need to have access to the local variables and parameters for this function call (which, as we know, are stored in the stack). This forces us to keep the function's **stack frame** in the **stack** until the function's work is done.

However, we could achieve the same result in a slightly different way. Instead of building the sum backward from each successive recursive call, we could **pass the partial sums forward** into the recursion - each successive call to the **recursive function** does all of its work **before the recursive call**, and thus it no longer needs to hang around in the **call stack**.

Let's have a look at how we could write the **sum()** function in that way:

```
sum_TR(array, n, part_sum)                      : L1
  if (n==0) return part_sum+array[0]            : L2
  else return sum_TR(array,n-1,part_sum+array[n])  : L3
```

This version is very similar to our original one but: It takes one extra parameter, **the partial sum** computed by previous calls to **sum_TR()**. The recursive case in **L3** is a bit different as well:

```
  else return sum_TR(array,n-1,part_sum+array[n])   : L3
```

In this version, there is **no computation** and **no work** done **after the recursive call** - once we call **sum_TR()** recursively, we are done and whatever the recursive call returns, that is the result we want. Because the **recursive call** is the **last thing the function does**, we call this a **tail-recursive** function.

Let's see how this may change the process when the **tail-recursive** version of the sum function is called with the same sample array as the original, recursive **sum()** function discussed earlier:

```
  sum_TR([10,5,3,2,1,8],5,0)                   // First call, n=5, part_sum=0  (A)

  // The call in (A) passes a partial result, array[5]+0 = 8 into the recursive call.
  // it's work is done so we do not need to keep it in the stack! memory for (A) is released

  sum_TR([10,5,3,2,1,8],4,8)                   // Second call, n=4, part_sum=8 (B)

  // The call in (B) passes a partial result, array[4]+8 = 9 into the recursive call.
  // (B) is done, we can release its memory from the stack

  sum_TR([10,5,3,2,1,8],3,9)                   // Third call, n=3, part_sum=9  (C)

  // (C) passes a partial result, array[3]+9 = 11 into the recursive call.
  // (C) is done, we release its memory from the stack

  sum_TR([10,5,3,2,1,8],2,11)                  // 4th call, n=2, part_sum=11   (D)

  // (D) passes a partial result, array[2]+11 = 14 into the recursive call
  // (D) is done, removed from the stack

  sum_TR([10,5,3,2,1,8],1,14)                  // 5th call, n=1, part_sum=11   (E)

  // (E) passes a partial result, array[1]+14 = 19 into the recursive call
  // (E) is done, removed from the stack

  sum_TR([10,5,3,2,1,8],0,19)                  // 6th call, n=0, part_sum=19   (F)

  // (F) reaches the base case, and returns array[0]+19 = 29, which is the final
  // result. (F) is done and removed from the stack, the final result is returned.
```

The process above is illustrated in Fig. 5.20.

With a small change to how our **recursive function** is implemented, we **eliminate** the need to keep multiple **stack frames** for the function within the **call stack**. This means the **tail-recursive** version can handle inputs that are much bigger (i.e. it works for much larger N) than the original **non-tail-recursive** version. Secondly, the result of the computation is available **as soon as we hit the base case**, no additional work is needed. The result of these two factors means that the **tail-recursive** version of the sum function is **as efficient** as a **iterative** (non-recursive, using for loops) function that computes the sum of the entries in an array.

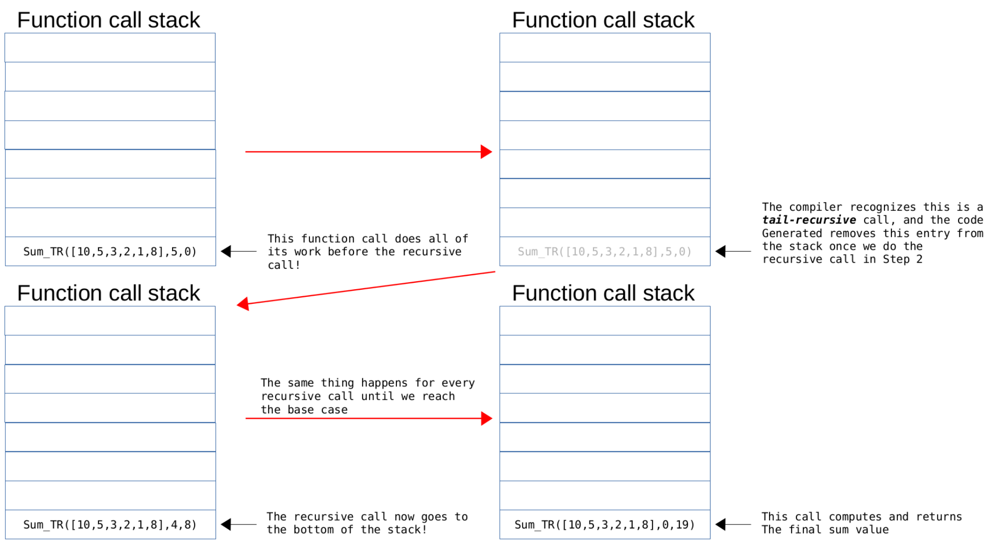

Function call stack
Sum_TR([10,5,3,2,1,8],5,0)
This function call does all of its work before the recursive call!
Function call stack
Sum_TR([10,5,3,2,1,8],5,0)
The compiler recognizes this is a tail-recursive call, and the code Generated removes this entry from the stack once we do the recursive call in Step 2
Function call stack
Sum_TR([10,5,3,2,1,8],4,8)
The same thing happens for every recursive call until we reach the base case
The recursive call now goes to the bottom of the stack!
Function call stack
Sum_TR([10,5,3,2,1,8],0,19)
This call computes and returns The final sum value

Figure 5.20: Illustration of how the **tail-recursive** function works at the **function call stack** level. Because each call **completes its work** before the recursive call takes place, it does not need to be kept in the stack. As a result there is never more than **one stack frame** reserved for the function in the **stack**, and in fact **the same stack frame** can be re-used by each successive call - thereby eliminating the **overhead** of **reserving and releasing space** from the **call stack**.

To test this, we can measure the time it takes to compute the sum of the entries in an array with $10,00$ floating point numbers using three different implementations of the sum function:

- **Iterative** sum - with for loops
- **Recursive** sum - the original **sum()**
- **Tail-recursive** sum - the **sum_TR()** we just developed

the sum is performed $100,000$ times with each function, and the total time is reported (doing the sum a single time is just too fast to accurately measure time taken on modern computers). The results are as follows:

```
./sum_runtime_analysis
For loops, sum=4988.303693, time taken=0.805763
Recursion, sum=4988.303693, time taken=1.389484
Tail recursion, sum=4988.303693, time taken=0.797839
```

We can quickly see that the **plain recursive** version of the sum function is about **70%** slower than the **iterative** function using only for-loops. We can also see that the **tail-recursive** version is **just as fast** - there is no visible **overhead** caused by the use of **recursion** (the tiny difference is not meaningful, and likely caused by timing inaccuracies in the computer where the program was run).

Note

Tail recursion optimization is a feature of modern programming languages and modern compilers. The compiler, or the interpreter for the language, must be able to recognize that a function is **tail-recursive**, and must be able to carry out the required optimizations to ensure no space is reserved in the **call stack** for successive calls to the **recursive function**, and that the result is returned directly from the last call when it reaches the **base case**. Put in a different way, if the compiler or language does not support **tail recursion optimization**, the two functions **sum()** and **sum_TR()** will behave in exactly the same way, use up a lot of space in the **stack**, and both would be just as slow when compared to the **iterative** version. Luckily, most current compilers and programming languages support this optimization.

So, in conclusion, when you are thinking of solving a problem with **recursion**, it is well worth the time to think of whether or not you can design your solution so that it is **tail-recursive**. It should be noted that **not all recursive functions can be made tail-recursive**. In such cases, the **overhead** of recursion must be carefully considered and accounted for.

5.7 Solving graph problems with recursion

We started this Chapter by learning about **graphs** and the kinds of problems we can solve with them. We said that **recursion** is a natural tool for working with **graphs** since graphs are recursive in nature. Now that we have learned about **recursion**, let's go back and solve the path-finding problem shown in Fig. 5.21. In this problem, we have a **start location** (the mouse), a **destination** (the cheese), and a **graph** that has $8 \times 8 = 64$ different locations. Each of these is represented by a **node**. To identify which **node** corresponds to **which** location, we can number the nodes starting with 0 at the **top-left** corner of the maze, and going all the way to 63 at the **bottom-right**.

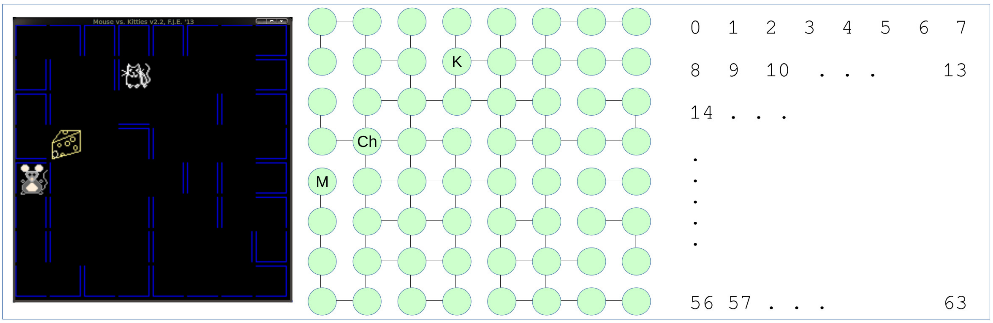


Figure 5.21: The problem of finding a path in a maze. Each location in the maze becomes a **node** in a **graph**, the **index** for the node corresponds to its **location** in the maze as shown.

We can represent the **connectivity** of the maze with the **edges** in the graph. If **two neighbouring locations** are **connected** (i.e. if the mouse can move from one to the next, because there is no wall separating them) we will have an **edge** joining the corresponding **nodes**. **Edge** information can be stored in an **adjacency matrix** or an **adjacency list**. For simplicity, here we will assume an **adjacency matrix** is used.

The problem we have to solve is finding a path, which consists of a sequence of connected nodes, leading from the start node to the destination. Let's formulate a **recursive** solution for this problem.

The base case: The **simplest** version of this problem occurs when **the start node** is the same as the **destination** node, in which case we are done. This is analogous to how a single-entry array is already sorted and there is nothing more to do. This is, however, not the only **base case** - what would happen if there was no path from **start** to **destination**? in that case we would expect to search through the entire graph and come up with no solution. So another **base case** is: there are no more **nodes** in the graph that we can **reach** from the **start node**.

The recursive case: This will deal with finding a path from **some starting** node (let's call it the **current node**) to the **destination**. It must reduce the size of the problem in some way so that eventually we reach one of our **base cases**. In the case of path finding, a common **recursive case**:

- Removes the **current node** from the graph (to create a smaller sub-graph).
- Recursively finds a path from **each neighbour of the current node** to the **destination** in the smaller sub-graph.
- If any of the neighbours returns a valid path, **build** a longer path from the **current node** to the **destination** by **pre-pending** the **current node** to the path returned by the neighbour.

Implicit in the above is that the **recursive case** will consider each **node** in the **graph** at most once (once it is removed by the **recursive case**, it will never again be reached from any of the sub-graphs).

The **order** in which we check for a path from the neighbours to the **destination** is important, it is called the **exploration strategy**. Different **strategies** will result in different search patterns (much like different ways for splitting an array result in different sorting methods). Depending on our choice of **strategy**, we may find completely different paths from one node to another. The choice of ordering strategy is a topic for an Artificial Intelligence book. There is no strategy that is **optimal for every graph search problem**, instead, different problems require different strategies.

For the purpose of understanding how we can apply **recursion** to solve path finding on **graphs**, let's look at the pseudo-code for the simplest strategy - one that is **purely recursive**. This exploration strategy is called **DFS** for **Depth-First Search** and it is a fundamental technique in graph search. **DFS** forms the basis of algorithms used for **scheduling** and other **constraint satisfaction** problems.

```
// Inputs: current              (index for the current node)
//        destination           (index for the destination node)
//        adjacency_matrix[][] (NxN array representing edges in the graph)
//        visited[]             (an array of size N, initially all zeros)

DFS(current, destination, adjacency_matrix, visited)

    visited[current]=1;          // Mark current node as 'visited' (this
                                 // removes it from the subgraph)

  if 'current' is the same as 'destination'

       return destination        // We got to our first base case

  otherwise

    for each neighbour of current, if visited[neighbour]==0

      subpath=DFS(neighbour, destination, adjacency_matrix, visited)
      if  'subpath' is not <empty>

         // prepend current to subpath to build full path
                return {current -> subpath}

       // We checked all neighbours and found no path...

    return (<empty>) // Second base case, no subpath from 'current'
                     // to 'destination' can be found
```

The **DFS** algorithm as shown above is well worth knowing. If is found in many application domains, and can be used to solve many general problems that require exploring a **graph**.

✍ **Exercise 5.7** Implement **DFS** for path finding on a grid of locations just as in Fig. 5.21. You don't need to actually create data for **nodes** in the graph! Since we are not actually doing any information processing at this point (only finding paths from one node to another by their index in the grid) it is enough to create an **adjacency matrix** (for a grid of size $m \times n$ the adjacency matrix has a size of $mn \times mn$) and then implement the **DFS** algorithm above.

Here is a sample **adjacency matrix** you can use to develop and test your **DFS** implementation. This is for a 4×4 grid:

```
int Adj[16][16]={{0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0}
    {0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0}
    {0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0}
    {0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0}
    {1,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0}
    {0,1,0,0,1,0,0,0,0,1,0,0,0,0,0,0}
    {0,0,1,0,0,0,0,1,0,0,1,0,0,0,0,0}
    {0,0,0,1,0,0,1,0,0,0,0,1,0,0,0,0}
    {0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0}
    {0,0,0,0,0,1,0,0,0,0,1,0,0,1,0,0}
    {0,0,0,0,0,0,1,0,0,1,0,1,0,0,1,0}
    {0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,1}
    {0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0}
    {0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0}
    {0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0}
    {0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0}};
```

Once you have completed the implementation of **DFS**, test it by finding paths between different pairs of nodes in the graph and checking that they make sense for the grid we have. This will require you to:

- Draw the graph as a grid of 4×4 nodes, linked as per the **adjacency matrix**
- Follow the paths returned by **DFS** for each of your tests, and verify they are valid (i.e. they visit neighbouring nodes that are connected in a sequence that leads from **start** to **destination)**

Note: The **DFS** process includes a step where the path is grown by pre-pending the **current node** to a **subpath** returned by a recursive call to **DFS**. This step requires thought. You're expanding a data structure whose size is not known in advance (you don't know the length of the path you will find). So think carefully about how to implement this, and remember we have several data structures we can use at this point which allow you to add data on-demand.

5.8 Debugging recursive code

Testing and **debugging** recursive code can be tricky. But with a bit of practice, and if you think carefully about what your recursion is supposed to be doing, you'll be able to handle any recursive function.

What we need to understand before we start debugging:

- The base cases - double-check that every base case has been considered, and that they are all in the implementation.
- The recursive case - check that the the splitting strategy is reasonable, and that it is implemented properly. Write down a step-by-step example of the entire process for an input that is small (so you can walk through the entire process by hand) but large enough to get the recursion going for a couple of levels at least.

Tracing your recursive code: The tricky part here is that **the same function gets called over-and-over-and-over**. So we **need additional information** in order to figure out what step of the **recursion** each call corresponds to, and then determine at which step something is not working right. How can we do this?

First find a small-enough case that fails: As we saw at the end of Chapter 4, **debugging** starts with a failed test. So it is important to find an input that is **small enough you can verify the process by hand** and for which the program produces an incorrect result. Once we have the test case as well as a hand-written step-by-step solution for that test case, choose whether we want to use a **debugger** or whether we want to use **print statements** to trace through the program.

Tracing using a debugger: If we are using a **debugger**, then we want to **insert breakpoints** at the **start of the recursive function**, so that once the recursive function begins we can **step through** the code in the function and compare what it is doing against our **hand-written solution** for the selected test case. The goal is to verify that at each step the function carries out the correct process, that the **recursive case** is correctly splitting the problem into sub-problems, and that the **base cases** are being triggered when found. We will need to have pen and paper, as we will need to keep track of **recursion depth** and possibly other information needed to indicate what step of the process we are looking at.

Tracing using printf(): The goal is the same as when we are using a **debugger**, only in this case it is not interactive. Information is printed out for us to examine after the program has run. Because the **recursion** involves a sequence of calls, there will likely be a sizable number of lines of text with information for us to walk through and compare with out hand-written solution. To make the process easier we can do **one or more** of the following:

- We can add a parameter to your function that indicates the **recursion depth** - that is, the step within the recursion to which any printed information for this particular call corresponds. The **recursion depth** starts at **zero** when we first call the recursive function, and thereafter is increased by **1** by each successive recursive call.
- We can **indent** the information printed based on the **recursion depth** - so there is a visual indication of what level within the recursion the printed data corresponds to.
- We can **dump the printed information to a log file** - because reading from the screen makes it harder to go back and forth as we are looking for a problem, and it is possible that if there is a large number of print

statements, the earlier ones may have been lost because the terminal keeps only a fixed number of lines of text.

- We can add a **pause** within the recursive function (at an appropriate place) to give us time to check the information that was printed out for each **recursive** call before continuing with the next one.

Note

Dumping the program's output into a log file is easily done:

```
// On Linux/Mac (from the terminal)
./my_program  >  log.txt

// On Windows
.\my_program.exe > log.txt

// The above will result in a new text file called 'log.txt'
// that contains the output of any printf() statements in
// 'my_program'
```

Adding a **pause** statement is easily accomplished by adding something like this:

```
printf("Press [ENTER] to continue...\n");
fgets(&dummy[0], 10, stdin);

// you must have declared a dummy string somewhere else in the
// function:
// char dummy[10];
```

this simply waits for the user to input some string. For the purpose of debugging, it waits for you to press **ENTER**. Notice that any text that is typed-in will be ignored.

Having a well organized **log** can make the difference between being able to quickly locate a problem, or spending a large amount of time trying to make sense of the information printed out in the log. The closer the structure of the printed output resembles the structure of the recursion (illustrated in Fig. 5.22) the easier it will be for us to figure out where any problems may be.

A couple common problems often found with code that uses recursion:

- It never reaches the **base case** (we can see the **depth** increasing to unreasonable values given the input) - we should check the **recursive case**, make sure it is making the problem smaller, and then check that the **base cases** have all been identified and implemented.
- It runs out of **stack space** for **recursive** calls. This can happen with large problems which require very deep sequences of recursive calls and use a large amount of memory for each call. Consider whether it would be possible to implement a **tail recursive** version of the function. It this is not feasible, we can use the **depth** variable to enforce a **maximum allowed recursion depth**. Our **recursive** call checks the depth value, and if it is equal to the **maximum allowed recursion depth**, it prints a message to let the user know the problem is too big to solve with the available stack memory. The advantage of doing this is that it prevents the program from crashing.
- The **recursion** doesn't return the correct solution - we should check that the **base case** is returning the correct

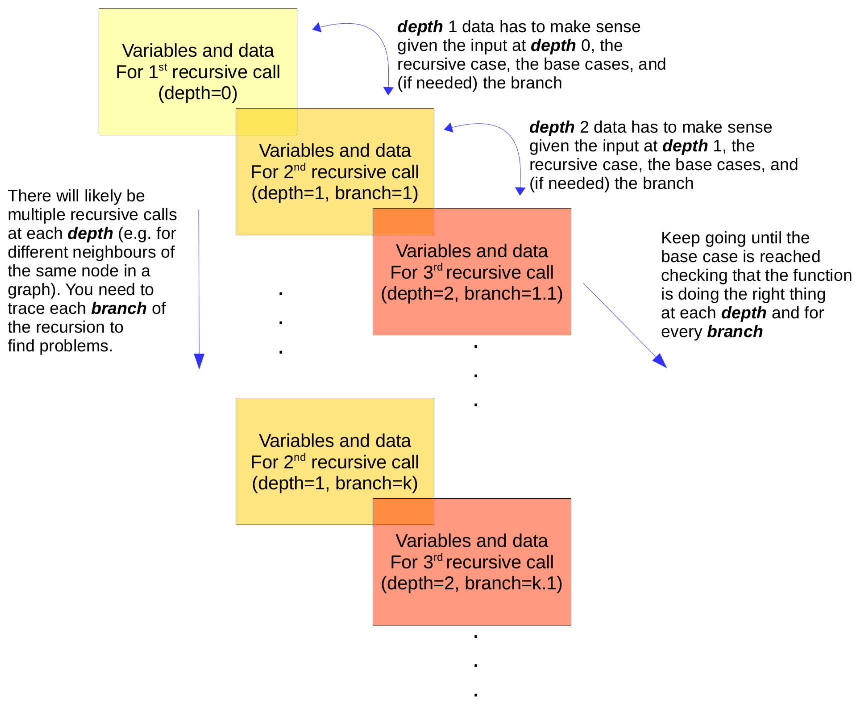

Variables and data For 1st recursive call (depth=0)
depth 1 data has to make sense given the input at depth 0, the recursive case, the base cases, and (if needed) the branch
Variables and data For 2nd recursive call (depth=1, branch=1)
depth 2 data has to make sense given the input at depth 1, the recursive case, the base cases, and (if needed) the branch
There will likely be multiple recursive calls at each depth (e.g. for different neighbours of the same node in a graph). You need to trace each branch of the recursion to find problems.
Variables and data For 3rd recursive call (depth=2, branch=1.1)
Keep going until the base case is reached checking that the function is doing the right thing at each depth and for every branch
Variables and data For 2nd recursive call (depth=1, branch=k)
Variables and data For 3rd recursive call (depth=2, branch=k.1)

Figure 5.22: Structure of the **recursive process** showing how data at different **depths** relates to previous recursive calls. Whatever tracing process we choose to use, we need to remain aware of what step in the recursion we are looking at, so we can check the information available to us against our hand-written solution.

result, and then check for correctness the process of building a bigger solution from the smaller one.

This concludes our study of **recursion**, one of the most important and useful tools for problem solving in computer science. It is a concept that is often in the minds of people who work with computers, so we close this Section with a couple of cartoons about recursion, courtesy of Randall Munroe of **XKCD** (`http://www.xkcd.com`).

5.9 Additional Exercises

The importance of choosing the right tool - We have claimed that you can solve any problem in either a **recursive** way or with loops. Choosing the right tool for a job matters. See if you can figure out how to solve the following problems using the tool indicated with each exercise.

✍ **Exercise 5.8** Assume we have a **BST** that stores unique **integer keys**. Write an algorithm that performs **in-order BST traversal** on this **BST** and prints the keys in sorted order. However, you are **not allowed to use recursion**. Use only loops to solve this task. You can use helper data structures as needed (hint, you will need to do some book-keeping that a recursive solution implicitly takes care of).

✍ **Exercise 5.9** **Flood-fill:** A common operation in paint programs is that of filling a closed region with colour. This is called **flood-fill** because we can think of it as pouring paint into the region, which then spreads out until it reaches the boundary of the region. The process is illustrated in Fig. 5.24.

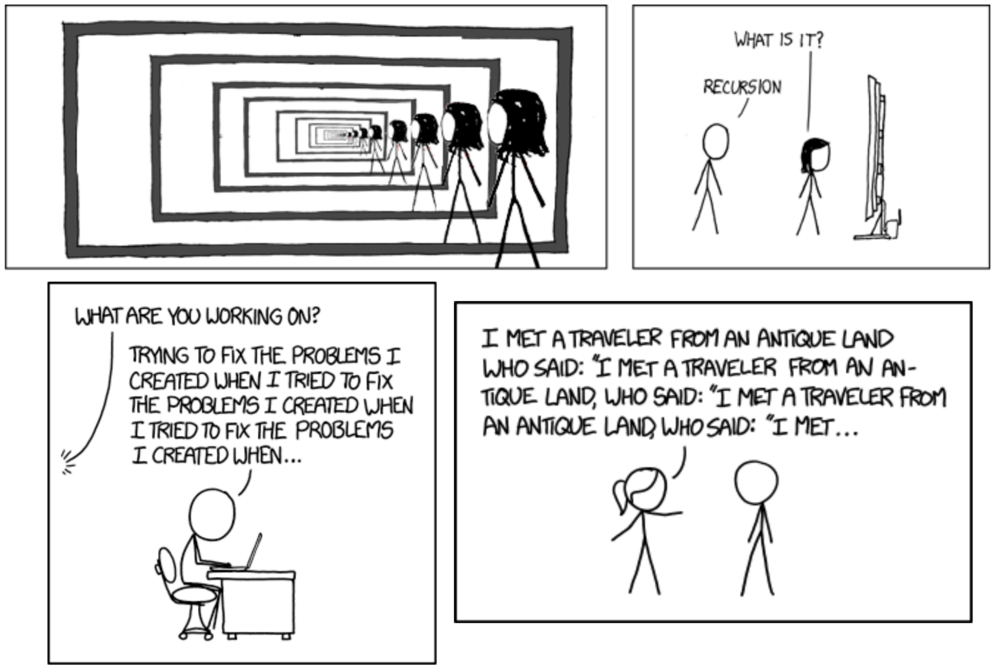

WHAT IS IT?
RECURSION
WHAT ARE YOU WORKING ON?
TRYING TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN...
I MET A TRAVELER FROM AN ANTIQUE LAND WHO SAID: "I MET A TRAVELER FROM AN ANTIQUE LAND, WHO SAID: "I MET A TRAVELER FROM AN ANTIQUE LAND, WHO SAID: "I MET...

Figure 5.23: A couple of cartoons about **recursion** from **XKCD**. *Courtesy of Randall Munroe, xkcd.com.*

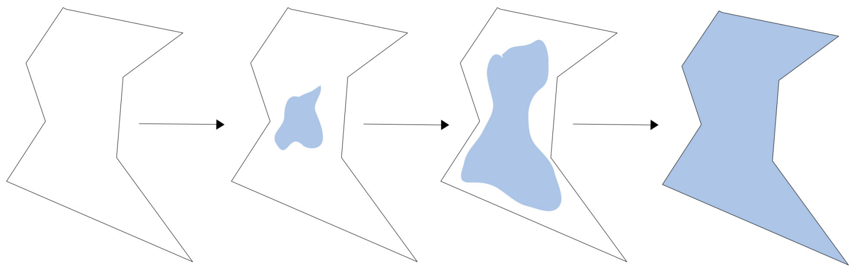


Figure 5.24: The **flood-fill** process simulates pouring paint into a closed shape. The paint spreads out until it reaches the boundaries of the shape. The process stops when the entire shape is filled with paint.

For the purpose of this exercise, the image will be a 2D array of size 500×500, and it will represent a **black and white** picture. Black pixels will have a value of **0** and white pixels will have a value of **1**.

- Write the algorithm for performing **flood-fill** on a closed shape such as shown in Fig. 5.24 using only loops, no recursion
- Write the algorithm for performing **flood-fill** using **recursion**.

Hint: For both cases, it helps to think of the process of paint spreading in order to figure out what the algorithm should do next.

✍ **Exercise 5.10** **Graph distance** - One common problem in **graph-based applications** is finding the subset of nodes that is reachable from a given **node** within a certain number of **hops** (i.e. going from one **node** to a neighbouring **node** is one **hop**). For example, suppose we have a **graph** representing the pre-requisite structure of courses at a University, and we want to know which courses have **Introduction to Cell Biology** as a pre-requisite, or as a pre-requisite to their pre-requisite. This would require us to find any **nodes** in the **pre-requisite graph** that are up to two hops away from **Introduction to Cell Biology**. Another example, in a **social network** representing people (as **nodes**) and their social connections **as edges**, we may ask for a list of people who are either **friends**, **friends-of-friends**, or **friends-of-friends-of-friends** of a particular person in order to invite everyone to a big party. This is equivalent to finding all the **nodes** in the social network **graph** that are at a distance of **3 or less** from the one representing the person whose friends we want to invite over.

The task: Given the adjacency matrix for the graph:

- Propose an algorithm that finds all **nodes** within a distance **k or fewer hops** from a selected **start node** using only loops.
- Propose an equivalent algorithm (i.e. one that solves exactly the same task) using **recursion**.

Question: Which of the two versions of the algorithm is more **intuitive** and **easier to understand and generalize**?

✍ **Exercise 5.11** **Crunchy!** - **DFS** is only one possible **strategy** for exploring a **graph**. A different method, called **breadth-first search (BFS)** uses a different order of exploration. For **DFS**, a node's grand-children, great-grand-children, and so on, will be explored before all of the node's children are explored. To see this, trace through the **DFS** pseudo code on a small graph (say, 4×4) and notice the order in which **nodes** are visited.

With **BFS** on the other hand, all the node's children are all explored first (before any grand-children). Then all of the grand-children are explored (before any great-grand-children), and so on until the entire graph is explored.

BFS is normally implemented using a queue (no need for **recursion**). Develop **pseudo code** for implementing the **BFS** algorithm for exploring a graph. Assume you have an implementation of a **queue** (you may want to review the operations supported by a **queue** in Chapter 3).

BFS is used for graph search, so it may help to think in terms of the path-finding problem described earlier in the chapter. You can assume there will be a **start node** and a **destination node**, and we want **BFS** to find and return a path from **start** to **destination**.

✍ **Exercise 5.12 Tail recursion:** - Implement a function that computes the **factorial** of an integer n **iteratively** (using only loops). Then implement a **recursive** version of the function. Finally, see if you can implement the **recursive** version so that it is **tail-recursive**.

Once you have the three different implementations, test them for **speed** by

- Writing a pair of **nested for loops**. The outer loop runs **100 iterations**, the inner loop runs **1,000,000** iterations. In the body of the inner loop there is a single instruction that computes the factorial of some number, e.g. 25, using **one** the three functions you wrote
- Compile and run the program, and time how long it takes to complete
- Change the function used to compute the factorial, and run the program again. Record the run-time for all three different implementations

Note: To measure the time it takes for a program to complete you can do the following on Linux and Mac:

```
>time my_program
```

this will print out the time taken by **my_program**

✍ **Exercise 5.13 Tail recursion:** - Implement a function that computes the n^{th} **Fibonacci** number using only loops. Then implement a **recursive** version of the function. Finally, implement a **tail-recursive** version of the function. Once you have the three different implementations, test them for **speed** as described in the previous exercise.

5.10 Building programs that work - Part 5

Through the previous chapters, we developed a process for **solving a problem**, **developing our solution into a program design**, **implementing and testing as we develop** to produce code that works, and **debugging** to find and resolve problems identified through testing or that show up once users are running our programs.

Now we should look at two very powerful tools to help us find and fix a wide variety of problems that result from bugs in how our program **uses or accesses memory**. This class of bugs tends to be particularly tricky to find and fix using standard tools such as **debuggers** or **print statements**. This is because often their behaviour is somewhat random, and may or may not be **triggered** by a specific test, or may occur under particular testing conditions which are hard to identify.

Luckily, there are tools designed to **check every memory access** our program does, and report any problems so we can fix them. These tools are:

- **valgrind**, a free **open-source** tool that has been the standard for memory checking and debugging on Linux and Mac for a long time. This can be installed directly from your computer's **package manager**.
- **Dr. Memory**, a newer, free **open-source** tool that is available for all platforms: Linux, Mac, and Windows, you can find it here: `https://drmemory.org/`.

Both carry out a thorough check of your program's memory accesses, and can help you find a large variety of bugs including very common ones such as:

- **Off-by-one errors** - and in general **index-out-of-bounds** errors when working with arrays.

- **Invalid memory access** - such as using a pointer and an offset to read or write into memory that is not reserved for the program.
- **Use of uninitialized variables** - which happens when we declare a variable, and then use it **before** we actually assign a value to it. As you remember, variables contain **junk** until we give them a value.
- **Use of uninitialized data** - similar to uninitialized variables, but happens when we use **malloc()** to get memory and then forget to fill that memory with valid data before using it in the program in some way.
- **Memory leaks** - which occur when we reserve memory with **malloc()** or **calloc()** but forget to release it with **free()** before the program ends.
- **Double free or invalid free** - which happens when we try to release memory more than once, or when we try to release memory that was not reserved with **malloc()** or **calloc()**.

Both of these tools offer a significant advantage when trying to resolve bugs related to memory access. But bear in mind that **both of these tools are complex** and they will require you to practice **using them** and **understanding the output**. There are extensive tutorials for both **valgrind** and **Dr. Memory**, and you should spend a bit of time learning how to use at least one of these.

But, to give you an idea of what these tools do, here is a short program with plenty of memory access problems:

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
  int array[10];
  int d;
  int *p;
  float *fp;

  // Fill the array with values
  // depending on if d=0 or
  // d=1 (unfortunately, it seems we
  // forgot to set d to some value!)
  if (d==0)                               // Problem #0
  {
    for (int i=0; i<10; i++)
      array[i]=i;
  }
  else
  {
    for (int i=0; i<10; i++)
      array[i]=i*i;
  }

  // Print the array, unfortunately
  // we have an off-by-one error
  // in the for loop!
  for (int i=0; i<=10; i++)               // Problem #1
   printf("array[%d]=%d\n",i,array[i]);

  // Get a pointer to the array and use
  // it to change some values,
  // unfortunately we are trying to access
  // values that are not in the array
  p=&array[0];
  *(p+250)=15;                            // Problem #2

  // Let's get some memory for floating point
  // data
  fp=(float *)malloc(5*sizeof(float)); // 5 floats requested

  // And the line below should crash the program
  // because we are trying to free memory for
  // an array that was not dynamically allocated!

  free(p);                                // Problem #3

  return 0;  // All good?

  // No! we forgot to free() the memory assigned to
  // fp, so there is a memory leak!

                                          // Problem #4
}
```

See what happens when we compile and run the program:

```
>./a.out
array[0]=0
array[1]=1
array[2]=2
array[3]=3
array[4]=4
array[5]=5
array[6]=6
array[7]=7
array[8]=8
array[9]=9
array[10]=1955339008
double free or corruption (out)
Aborted (core dumped)
```

Obviously something went wrong, so we need to **trace and debug** the program - in the example above it is easy because it is short and we already know what the problems are, but you can imagine in a large, complex piece of software finding a memory access issue will be difficult. Let's see what we can learn by running **Dr. Memory** and **valgrind** on the program above. The output from **Dr. Memory** is shown below:

```
> ./drmemory -- ./a.out
~~Dr.M~~ Dr. Memory version 2.6.0
~~Dr.M~~ WARNING: application is missing line number information.
~~Dr.M~~
~~Dr.M~~ Error #1: UNINITIALIZED READ: reading 0x00007ffeffd5495c-0x00007ffeffd54960 4 byte(s)
~~Dr.M~~ # 0 main
~~Dr.M~~ Note: @0:00:00.297 in thread 5316
~~Dr.M~~ Note: instruction: cmp 0xffffffbc(%rbp) $0x00000000
array[0]=0
array[1]=1
array[2]=2
array[3]=3
array[4]=4
array[5]=5
array[6]=6
```

```
array[7]=7
array[8]=8
array[9]=9
array[10]=-1084233984
~~Dr.M~~
~~Dr.M~~ Error #2: INVALID HEAP ARGUMENT to free 0x00007ffeffd54970
~~Dr.M~~ # 0 replace_free           [/home/runner/work/drmemory/drmemory/common/alloc_replace.c
    :2710]
~~Dr.M~~ # 1 main
~~Dr.M~~ Note: @0:00:00.324 in thread 5316
~~Dr.M~~
~~Dr.M~~ Error #3: LEAK 20 direct bytes 0x00007f8cec2f13a0-0x00007f8cec2f13b4 + 0 indirect
    bytes
~~Dr.M~~ # 0 replace_malloc           [/home/runner/work/drmemory/drmemory/common/alloc_replace
    .c:2580]
~~Dr.M~~ # 1 main
~~Dr.M~~
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~      0 unique,    0 total unaddressable access(es)
~~Dr.M~~      1 unique,    1 total uninitialized access(es)
~~Dr.M~~      1 unique,    1 total invalid heap argument(s)
~~Dr.M~~      0 unique,    0 total warning(s)
~~Dr.M~~      1 unique,    1 total,    20 byte(s) of leak(s)
~~Dr.M~~      0 unique,    0 total,     0 byte(s) of possible leak(s)
~~Dr.M~~ ERRORS IGNORED:
~~Dr.M~~     14 unique,   17 total,  7610 byte(s) of still-reachable allocation(s)
~~Dr.M~~         (re-run with "-show_reachable" for details)
~~Dr.M~~ Details: /home/strider/mtvp/DrMemory-Linux-2.6.0/drmemory/logs/DrMemory-a.out
    .5316.000/results.txt
```

Notice the line

```
~~Dr.M~~ Error #1: UNINITIALIZED READ: reading 0x00007ffeffd5495c-0x00007ffeffd54960 4 byte(s)
```

It indicates we are accessing **uninitialized memory** in the program. **it is not clear exactly where** this happens because **Dr. Memory** is working on the **executable program** and can't reference your program's code.

In order to help understand what memory debugging tools provide us, **it helps to use print statements to tell us which part of the program is running**. We are limited to using print statements because we **can not run a debugger together with a memory checking tool**. So - let us change our program and add a few print statements:

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
  int array[10];
  int d;
  int *p;
  float *fp;

  // Fill the array with values
  // depending on if d=0 or
  // d=1 (unfortunately, it seems we
  // forgot to set d to some value!)

  printf("Initializing integer array... \n");
  if (d==0)                                  // Problem #0
  {
    for (int i=0; i<10; i++)
      array[i]=i;
  }
  else
  {
    for (int i=0; i<10; i++)
      array[i]=i*i;
  }

  // Print the array, unfortunately
  // we have an off-by-one error
  // in the for loop!

  printf("Printing the contents of the integer array... \n");
  for (int i=0; i<=10; i++)                 // Problem #1
   printf("array[%d]=%d\n",i,array[i]);

  // Get a pointer to the array and use
  // it to change some values,
  // unfortunately we are trying to acces
  // values that are not in the array
  printf("Using pointers to access array data... \n");
  p=&array[0];
  *(p+250)=15;                              // Problem #2

  // Let's get some memory for floating point
  // data
  printf("Reserving dynamic memory... \n");
  fp=(float *)malloc(5*sizeof(float)); // 5 floats requested

  // And the line below should crash the program
  // because we are trying to free memory for
  // an array that was not dynamically allocated!

  printf("Releasing memory before the program exits... \n");
  free(p);                                  // Problem #3

  return 0;  // All good?

  // No! we forgot to free() the memory assigned to
  // fp, so there is a memory leak!

                                            // Problem #4
}
```

Now let's re-compile the program and run **Dr. Memory** again, which results in the following output:

```
>./drmemory -- ./a.out
~~Dr.M~~ Dr. Memory version 2.6.0
Initializing integer array...
~~Dr.M~~ WARNING: application is missing line number information.
~~Dr.M~~
~~Dr.M~~ Error #1: UNINITIALIZED READ: reading 0x00007ffcce72498c-0x00007ffcce724990 4 byte(s)
~~Dr.M~~ # 0 main
~~Dr.M~~ Note: @0:00:00.272 in thread 5502
~~Dr.M~~ Note: instruction: cmp 0xfffffffbc(%rbp) $0x00000000
Printing the contents of the integer array...
array[0]=0
array[1]=1
```

```
array[2]=2
array[3]=3
array[4]=4
array[5]=5
array[6]=6
array[7]=7
array[8]=8
array[9]=9
array[10]=724062720
Using pointers to access array data...
Reserving dynamic memory...
Releasing memory before the program exits...
~~Dr.M~~
~~Dr.M~~ Error #2: INVALID HEAP ARGUMENT to free 0x00007ffcce7249a0
~~Dr.M~~ # 0 replace_free           [/home/runner/work/drmemory/drmemory/common/alloc_replace.c
    :2710]
~~Dr.M~~ # 1 main
~~Dr.M~~ Note: @0:00:00.295 in thread 5502
~~Dr.M~~
~~Dr.M~~ Error #3: LEAK 20 direct bytes 0x00007f0c80e213a0-0x00007f0c80e213b4 + 0 indirect
    bytes
~~Dr.M~~ # 0 replace_malloc           [/home/runner/work/drmemory/drmemory/common/alloc_replace
    .c:2580]
~~Dr.M~~ # 1 main
~~Dr.M~~
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~      0 unique,    0 total unaddressable access(es)
~~Dr.M~~      1 unique,    1 total uninitialized access(es)
~~Dr.M~~      1 unique,    1 total invalid heap argument(s)
~~Dr.M~~      0 unique,    0 total warning(s)
~~Dr.M~~      1 unique,    1 total,    20 byte(s) of leak(s)
~~Dr.M~~      0 unique,    0 total,     0 byte(s) of possible leak(s)
~~Dr.M~~ ERRORS IGNORED:
~~Dr.M~~     14 unique,   17 total,  7610 byte(s) of still-reachable allocation(s)
~~Dr.M~~         (re-run with "-show_reachable" for details)
~~Dr.M~~ Details: /home/strider/mtvp/DrMemory-Linux-2.6.0/drmemory/logs/DrMemory-a.out
    .5502.000/results.txt
```

Now we can see that **Error #1** occurred while **initializing integer array**, so it must be reporting the use of **d**, which was declared but not initialized (this is labeled **problem #0** in the program listing).

Interestingly, **Dr. Memory** did not catch the **off-by-one** error printing the array, or the **invalid pointer access** - this is likely because whatever memory locations were accessed are still within the program's reserved memory.

The next error to be reported, **Error #2** happens when **releasing memory before the program exits...**, and it states that an invalid argument was passed to **free()**. This corresponds to **problem #3** in the program listing.

Finally, **Dr. Memory** reports in **Error #3** that there were **20 bytes** of **memory leaks** (**problem #4** in the listing) corresponding to the 5 floating point values we allocated with **malloc()** and assigned to pointer **fp** but forgot to release.

All things considered, by using a combination of **carefully placed print statements** together with **Dr. Memory**, we were able to quickly identify 3 out of 5 problems in the program above. The remaining two would require **specific tests that cause the program to fail** to be run, together with the **print statement + memory checking** combination.

Let’s see what **valgrind** does with the same program (**valgrind** prints a lot of information messages that are not relevant for debugging, the output below contains only the lines pertaining problems with the program):

```
> valgrind --verbose ./a.out
==5911== Memcheck, a memory error detector
==5911== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5911== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==5911== Command: ./a.out
==5911==
--5911-- Valgrind options:
--5911--   --verbose

.
.
.

--5911-- REDIR: 0x4ed5020 (libc.so.6:malloc) redirected to 0x4c31aa0 (malloc)
Initializing integer array...
==5911== Conditional jump or move depends on uninitialised value(s)
==5911==   at 0x1087B1: main (in /home/strider/CS/course_material/MyCourses/Sabbatical_23-24/
    ICS_BOOK/scratch/a.out)
==5911==
--5911-- REDIR: 0x4fcc970 (libc.so.6:__mempcpy_avx_unaligned_erms) redirected to 0x4c39130 (
    mempcpy)
Printing the contents of the integer array...
--5911-- REDIR: 0x4fcc090 (libc.so.6:__strchrnul_avx2) redirected to 0x4c39020 (strchrnul)
array[0]=0
array[1]=1
array[2]=2
array[3]=3
array[4]=4
array[5]=5
array[6]=6
array[7]=7
array[8]=8
array[9]=9
array[10]=1001336320
Using pointers to access array data...
Reserving dynamic memory...
Releasing memory before the program exits...
--5911-- REDIR: 0x4ed5910 (libc.so.6:free) redirected to 0x4c32cd0 (free)
==5911== Invalid free() / delete / delete[] / realloc()
==5911==   at 0x4C32D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==5911==   by 0x10888C: main (in /home/strider/CS/course_material/MyCourses/Sabbatical_23-24/
    ICS_BOOK/scratch/a.out)
==5911== Address 0x1ffefffb50 is on thread 1's stack
==5911== in frame #1, created by main (???:)

.
.
.

==5911==
==5911== LEAK SUMMARY:
==5911==   definitely lost: 20 bytes in 1 blocks
==5911==   indirectly lost: 0 bytes in 0 blocks
==5911==     possibly lost: 0 bytes in 0 blocks
==5911==   still reachable: 0 bytes in 0 blocks
==5911==        suppressed: 0 bytes in 0 blocks
==5911== Rerun with --leak-check=full to see details of leaked memory
```

```
==5911==
```

Similar to **Dr. Memory**, **valgrind** finds:

- Conditional jump or move depends on uninitalized value(s) - problem #0
- Invalid free() - problem #3
- Memory leaks 'definitely lost: 20 bytes in 1 block' - problem #4

Also just like **Dr. Memory**, **valgrind** didn't catch the **off-by-one** array access problem, or the **invalid pointer access**. However, any test case where your program crashes will be caught by either of these tools, and the possible cause will be reported.

In conclusion: You should learn to use, and then always check your programs with a **memory checking tool**. It will likely reveal problems that were not visible during your testing and that would go undetected otherwise. While the tools aren't perfect and will not catch every single problem with a program, used in conjunction with **a thorough and well designed testing strategy** they will enable you to produce code that is sound and correct in terms of how your programs access information within the computer's memory. Spending time learning how to use a **memory checking tool** will turn you into a much more capable software developer, and is well worth the effort. Another good habit to develop, and a valuable skill to add to your toolbox.

Chapter 6 Designing and Building Good Software

So far, we have been thinking of problem solving in terms of the concepts, ideas, and algorithms we may need to use to get a particular task done. We should also spend some time considering the related issue of **how to organize the software** we are writing in a way that makes our solution as **useful to others** as possible.

In this Chapter, we will discuss the general principles that guide the **organization of larger pieces of software**. The goal is to understand **how and why** we need to think very carefully about the way we design the software we are writing, and to learn about the principles that have been developed to help us better organize, maintain, and expand on a given software solution.

The principles we will learn in this Chapter will be the foundation on which we will build a full understanding of the software design process, a topic you can explore in depth by reading up on software design, software engineering, and software architecture.

6.1 Software as a collection of modules

As we have worked our way through previous chapters, we have already been taking advantage of the significant amount of work done by others on our behalf. All the functions in standard **C** libraries that we have used to implement the different algorithms discussed in the book had to be **designed, developed**, and **tested** well before they shipped with your compiler so you could easily have things like **printf()** or **qsort()** at your disposal.

Question: what would software look like if we didn't have any libraries for commonly used functions? If we think about it for a moment, we will realize that such software would look like the illustration in Fig. 6.1.

THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

Figure 6.1: Software as a disorganized collection of **stuff** that does **something**. **Courtesy of Randall Munroe, http://www.xkcd.com**.

Without **some form of organization**, our software would be a big pile of code, with all kinds of functionality mixed in - there would be functions for carrying out the algorithms we need, but also all kinds of unrelated things like printing to the terminal, reading user input, managing files, implementing basic math, etc.

Anyone trying to understand our software, or worse, someone with the task of finding and fixing any existing **bugs** in our software would have a very difficult time dealing with our code.

So instead of writing large piles of code that does every kind of thing, we should instead **figure out how to break down our application** into a set of **separate components** each of which can be implemented as a **module** with its own code, and which can be **understood, tested**, and **debugged** without having to sift through the rest of the application's components (or with only minimal need to do so). As an additional but very important benefit - these modules can be **easily reused**, which means that they can be either directly combined with entirely different applications we are developing, or re-used with only minimal changes.

In effect, what we want is to build a **large library of modules** that we can **reuse with ease** to build any software we want. For example - we may want to write a module that provides support for creating, storing, managing, and running standard algorithms on **graphs**. It could, for instance, provide path-finding using **DFS**, allow us to modify the **graph** by adding or removing **edges** and **nodes**, and handle all the required data structures so that we don't have to re-write all of that code whenever we need to use a **graph**. We want to write this **module** in such a way that anyone else writing a program that needs to do path finding can easily take our module for handling **graphs** and use it without **needing to read through** our code in detail, and without needing to worry about **our code being full of bugs** or **being unnecessarily inefficient** (in terms of complexity).

This is not a trivial problem - out software needs to be **designed carefully and correctly**, so that it can work on a **graph** representing any data a developer may need to handle, so that it can easily be called from software not written by us, and so that the resulting programs can be **tested, debugged**, and possibly **expanded** by other developers who don't know (and don't need to know) the details of how we developed the **graph module** or even how the algorithms contained in the module work.

Note

If you think back to Chapter 3, we already have discussed this idea. Back then we said that **ADTs** are useful because they provide a developer with a concise idea of how the particular **ADT organizes data** and what **operations** it supports in a way that is **independent from implementation details**. This allows a developer to use **any implementation of the ADT** in their programs at the level of **data and operations on data** without needing to know how these are actually implemented.
With well designed **software modules** we are pursuing a similar result: We want to design **modules** that can be used in terms of **data** being stored and manipulated, and a set of **operations and algorithms** that can be applied to the data managed by the **module**. We want a developer to be able to use the **module** without needing to know or understand how the actual **operations and algorithms** are implemented, as this will free them to focus on their actual task and allow them to build sophisticated software with advanced functionality quickly, reliably, and without being experts on every possible aspect of computer science that may be involved in solving a particular problem.

Let us look at some of the **key principles** that should be in our mind when we start exploring the world of software design.

6.2 A wish list for building good software

If we are going to put our time and effort into developing software for solving a problem, we want to make sure the effort we put into it results in the best possible software. Below is a list of properties that we should carefully consider when designing our software - not every single one will apply to every single problem, but we must always consider them all before we sit down to design and develop our solution.

1) Modularity: Our software is composed of separate **modules** each of which has **one specific task**, and each of which is **self-contained**, so it can be **understood, tested**, and **maintained** independently of the others. A well designed modular program will help:

- Reduce **replication** of code
- Improve the chance that code we write will be **reused**
- Make it **easier to test** the code and verify it is correct
- Help a developer focus on **the big picture** of how a particular software is structured and how it works

2) Reusability: Writing good software requires a lot of thought and hard work, we want to ensure any **modules** we write for a specific task can be **re-used** by any other application that requires the same functionality . For instance, if we develop a **module** that implements shortest-path finding between two nodes in a graph (which, as we saw, is a common problem in many application domains). We want our implementation to be such that anyone needing to find the shortest path between graph nodes can simply take our code and use it to build their application.

3) Extendibility: We want our software to be **easy to extend and improve**. This allows us to build better, more capable software over time by improving and expanding its functionality.

4) Maintainability: Our software must be **well organized**, **easy to understand**, **well documented**, and **free of unnecessary complexity**. This improves our ability to **test it**, **debug it**, and **upgrade it** as needed. A competent developer not familiar with our code should be able to quickly get to the point where they can work on/with it.

5) Correctness: Any software we develop and release must have been **thoroughly tested** and made as close to **bug-free** as possible. Where appropriate, suitable tools should be used to determine **correctness**. Our code should have been **reviewed** by experienced developers not related to its implementation, and a suitable process must be in place for **documenting**, **tracking**, and **resolving bugs** found after the software is released.

6) Efficiency: We have spent a good amount of time thinking about **complexity** and how to study the efficiency of our algorithms. We expect good code to be **efficient** both in terms of the algorithm chosen to solve a problem, and also in terms of how that algorithm is implemented.

7) Openness: When possible (e.g. when we're not developing software for a company that has a claim of ownership over the code they pay us to write), we should consider contributing our work to the **open source software (OSS)** community. There is a lot of good work already out there that is done for no other gain than to provide **something useful for others**. And we can contribute to this effort. Open source software projects are a good way to make sure your work directly benefits others. We will get back to this at the end of the Chapter, in the last section on how to build software that works.

8) Privacy and security: A significant portion, or even the majority of the software we write will be handling potentially sensitive information. The data the program works with should be **protected** from **unauthorized access, use, or distribution**. Consideration must be carefully given to the **privacy of a user's personal information**, and if the software is **accessible over a network** then we have to follow best known practices for **securing access** to the software, **protecting information from falling in the hands of hackers**, and **ensuring no unauthorized copies** of any information managed by the application can be made or distributed without proper authorization. Very importantly **the user must always grant informed consent** before any personal information about them can be requested and stored by our application. This means **clearly and transparently** indicating **what information will be requested and stored**, **how the information will be used**, **how long it will be stored**, and **how (if at all) it will be shared** with any third parties. **No information should be gathered without informed consent**.

The above is not a comprehensive list. Specific applications will require additional consideration to be given to factors such as **reliability**, **failure-tolerance**, **computational performance**, **data storage requirements**, and many other possible factors. However, the properties listed above are a solid starting point on which we can build good software and apply to the vast majority of applications we may need to develop.

6.3 How modules are organized and used in C

So far, we have been working with applications that contain only a couple program files at the most, and we have not needed to break up our application into multiple independent **modules**. This is only suitable for small programs, and for applications with limited functionality. As we said above, we want to split complex applications into **modules** that implement part of the application's functionality. Each module will have its own program file(s), and they all need to be brought together in the right way to generate the executable program for our application.

In **C**, this is done by splitting each module into:

- **A header file:** These are files with extension **.h**, and contain only the function declarations, without any of the code. **Header files** also provide definitions for common things such as for instance **mathematical constants**. You have already been using header files for common **C** libraries such as **stdio.h**, and **stdlib.h**. These provide the function definitions the compiler needs in order to know what to do when you call functions such as **printf()** or **malloc()** that are provided by these libraries.
- **One of more program files:** These have the extension **.c** and contain the actual code needed to implement the functions declared in the header file. To create a working **executable program** we eventually need to have access to the implementation of **every function** that is used by the application, regardless of whether it is part of a library, or whether it is part of the code we have written ourselves.

Applications are built from multiple such modules by

- Compiling each separate **module** into **executable code** with **placeholders** for functions from other modules.

- **Linking** all the modules together: this means bringing together the **executable code** that implements the functionality from each of the modules being used, and **updating the place-holders with the actual function calls** to each implemented function's code.

The entire process is illustrated in Fig. 6.2. An application consisting of four different **modules** is split into multiple files - there are four **header (.h)** files, one for each module. There are also four **implementation (.c)** files with the corresponding implementation. Note that each **module** may use functions from the others, and uses **#include** statements to import the function declarations for those modules during compilation.

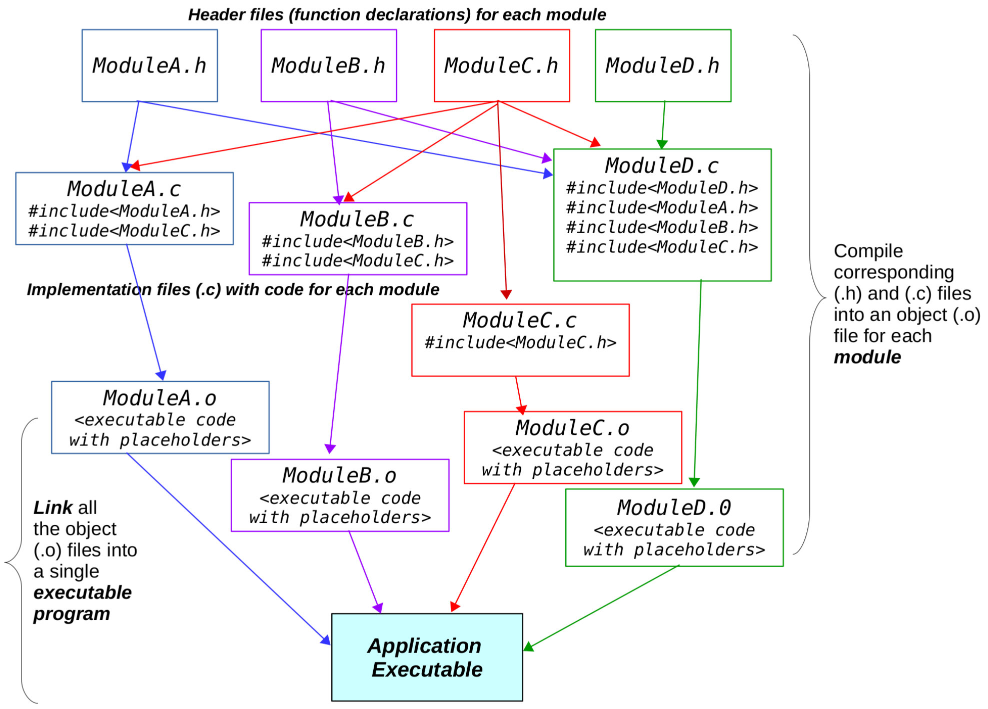

Header files (function declarations) for each module
ModuleA.h
ModuleB.h
ModuleC.h
ModuleD.h
ModuleA.c
#include<ModuleA.h>
#include<ModuleC.h>
ModuleB.c
#include<ModuleB.h>
#include<ModuleC.h>
ModuleD.c
#include<ModuleD.h>
#include<ModuleA.h>
#include<ModuleB.h>
#include<ModuleC.h>
Implementation files (.c) with code for each module
ModuleC.c
#include<ModuleC.h>
Compile corresponding (.h) and (.c) files into an object (.o) file for each module
ModuleA.o
<executable code with placeholders>
ModuleB.o
<executable code with placeholders>
ModuleC.o
<executable code with placeholders>
ModuleD.0
<executable code with placeholders>
Link all the object (.o) files into a single executable program
Application Executable

Figure 6.2: The process for combining multiple **modules** into a single **executable program**.

Each **module** is compiled into an **object (.o)** file that contains **executable code** for that particular module's functions. This file contains **placeholders** wherever functions from other modules are used. Once all the modules have been **compiled into object files**, all the **.o** files are **linked together**. This process involves combining the **implementation of any functions used in the program into a single executable file** and replacing the **placeholders** with actual working function calls. The result is a single, working, executable program.

All the **C** programs you have implemented and compiled up to this point had to go through this process. We did not have to think about it because since we have only used standard **C** libraries, the entire process was **done automatically and transparently**. However, once we start developing our own **modules** and developing applications consisting of multiple components, we will need to keep in mind how the different parts work together to build our application.

6.4 Interacting with modules: The Application Programming Interface (API)

Back to the problem of how to build our software properly. The key problem that concern us is **how will other programs interact with our module?** that means we have to think about the **functionality** that will be **provided by our module**, which usually means the set of **functions** other programs can call, and the way in which information will be **passed to and from** between other programs and our module's functions.

Setting down the details of how modules communicate and interact with each other is the job of the **Application Programming Interface** or **API**. The **API** contains all the specifications needed to make use of, and to interact with different **software modules**, **application libraries**, **local or remote computer systems**, and even **internet based applications**.

Note

Here are some examples of commonly used **APIs** you may need to work with in the future:

- **Google Maps:** Allows us to make use of the **Google maps** framework for plotting locations and for finding paths between points in maps - among many other things. You can check out the **API** at `https://developers.google.com/maps/documentation/javascript/tutorial`
- **TensorFlow:** Allows us to set up, train, test, evaluate, and operate machine learning algorithms, including **neural networks** for solving a task that require learning from a very large dataset. Nowadays such algorithms are behind some of the most useful applications in A.I. including **Large Language Models (LLMs)**. The **API** can be found at `https://www.tensorflow.org/`
- **Amazon AWS:** Amazon's cloud-based **AWS** runs a large portion of internet-hosted services, and powers all kinds of applications from on-line trade to providing computing power for large simulations. The **API** can be found at `https://docs.aws.amazon.com/index.html#lang/en_us`
- **Unity:** Possibly the most popular **API** for creating, manipulating, and rendering 3D content; from graphical user interfaces and simulations, to interactive programs and games. The **API** is at `https://docs.unity3d.com/ScriptReference/`

The above is just a tiny sample of the universe of **APIs** out there. Each of them is a world of complexity but the key is - we **don't have to ever look at the code that implements any of the functionality they provide** if we don't want to.

The usefulness of an **API** is that it allows us to use a **module, library**, or **service**, without having to know the details of how it's implemented. All we need to know is how to **pass information** to the functions in that module, and how to **get back results**. This will be easier or harder, depending on whether the **API** is well designed or not. A **badly designed API** will make software building **cumbersome** and **reduce the usability** of the module for which the **API** was designed.

In **C**, the **API** consists of the **function declarations** (in the **.h**) file, along with any **constants and other important values** defined there - it also includes all the **documentation** (at the top of each function) that describes **what each of the functions does** and their **parameters and return values**. In addition to this (but not necessarily required), there often exists some form of externally maintained **documentation** (e.g. manual pages, a wiki, or a webpage) that describes and summarizes the **API**, and often also provides examples of how to use it.

Let's now see what goes into designing a good **API**, and why it is a challenging but important process worth a significant amount of thought and care.

6.4.1 Why thinking carefully about API design matters

Suppose we are writing a module for graph manipulation that supports finding a path between nodes in the graph. The algorithm is implemented by this function:

```
intList *findPath(int Adj[][], int start, int goal)
{
   /*
      This function returns a linked-list of nodes that from a path
      from:
              start
      to:
              goal

  If no path can be found, the function returns NULL.

      Adj[][] is the adjacency matrix for a graph with size N

   Assumes: Adj(i,j) is 1 if there is an edge from i to j, and 0
      otherwise. i,j are node indexes in [0,N-1]

   intList is a linked list of nodes, each of which has:
      int nodeIndex;
      intList *next;

}
```

We don't need to know how the function works. All that matters is that the **function declaration** and the **comments** tell the developer that this function will return the path between **start** and **goal**, that the **path** will be in the form of a linked list of node indices, and that it describes the **input parameters** the function requires in order to do its work.

Thereafter, if we need to use that function in our program all we need to do is:

- Set up an adjacency matrix for my graph with size $N \times N$
- Find the **indexes** of the **start** and **goal** nodes
- Call the function

However, while considering this very straightforward **use case**, we will notice that our **API** is not designed properly - it does not provide a way for us to specify N, the size of the graph. So we need to change our API a bit:

```
   intList *findPath(int Adj[N][N], int N, int start, int goal);
```

With the above change, it seems our API is good to go and we can release this little function for other developers to use. However, as soon as we release the API, another developer comes along who wants to use our function but their **use case** is somewhat different. They need to find a path from a **start** node to **one of a number of possible goal** nodes (for example, to find a path from a current location to any nearby gas station while driving). With the current **API**, the developer is stuck doing something like this:

```
Set up an array of possible goal locations
for each location j in the goals array
    path = findPath(A,N,s,goals[j])
    if path is not NULL break
```

Which is not too bad, but this seems like a common-enough **use case** that we may want to provide the **API** with a way to to this, so we now re-define our **API** as follows:

```
intList *findPath(int Adj[][], int N, int start, int goals[k], int k)
{
   /*
      This function returns a linked-list of nodes that from a path
      from:
              start
      to:
              the *first* goal node in goals[] that can be
              reached during search.

  If no path can be found, the function returns NULL.

      Adj[][] is the adjacency matrix for a graph with size N

      goals[] is an array of integers with the index of any
          goal nodes that should be considered by the
          function.

      k is the number of entries in goals[]

   Assumes: Adj(i,j) is 1 if there is an edge from i to j, and 0
      otherwise. i,j are node indexes in [0,N-1]

   intList is a linked list of nodes, each of which has:
      int nodeIndex;
      intList *next;
   */
}
```

Question: Is the above actually better? is the latest version of the **API** more useful? is it **easier** to use and more **general**? As it happens, the answer may not be straightforward, and likely **there won't be an answer that suits every developer who wants to use this module**.

Example 6.1 After the module is released on **GitHub**, one of the developers who would like to use the module reports that their graph is **too big** and the **adjacency matrix doesn't fit in memory**. They would like to have a way to use an **adjacency list** instead.

That sounds reasonable, but we can not update the function call that uses an **adjacency matrix** to also work with an **adjacency list**. This is a problem we will return to again soon, we can solve it, but in **C** it would be pretty ugly, so we should solve it with a more advanced language.

For now, we decide we can help developers who have an **adjacency list** by adding one more function to the **API**:

```
  intList *findPath_l(intList* A[], int N, int start, int goals[k], int k)
  {
      This function returns a linked-list of nodes that from a path
      from:
              start
      to:
              the *first* goal node in goals[] that can be
              reached during search.

   If no path can be found, the function returns NULL.

      A[] is the adjacency list for a graph with size N

      goals[] is an array of integers with the index of any
         goal nodes that should be considered by the
         function.

      k is the number of entries in goals[]

    intList is a linked list of nodes, each of which has:
      int nodeIndex;
      intList *next;
   }
```

The example above illustrates a common problem when designing **APIs**. Often there are many **small variations** on a problem that users of a **module** or **library** may need support for - depending on their specific task and the rest of their program. However, adding **functions that are small variations of each other** is not a good solution: it creates a lot of **code duplication** since the functions have the same task, but they work on slight variations of the input which means a lot of the code in each function will be identical and yet not easily separable into smaller, independent functions. It also makes the **API** cumbersome since developers now have to worry about figuring out which among the many similar functions they should be using.

Consider what happens if a bug is found in **findPath()** - because there are two versions of it, we will have to fix a problem in two places. The more variants of **findPath()** that we provide for the convenience of developers using our **module**, the more places we will likely need to check and possibly update when problems are found. This makes our **module** more difficult to **maintain** and increases the likelihood we will miss something that needs updating. So it is far from an ideal situation.

This brings us to what happens whenever the **API** is updated. For instance, because **bugs were fixed**. Since this doesn't change any of the function declarations for the **module**, the change doesn't affect how programs using our module are written or how they interact with our **module**. However, in order for the update to take place on programs that use our **module**, every program using it must be **recompiled**. This is illustrated in Fig. 6.3.

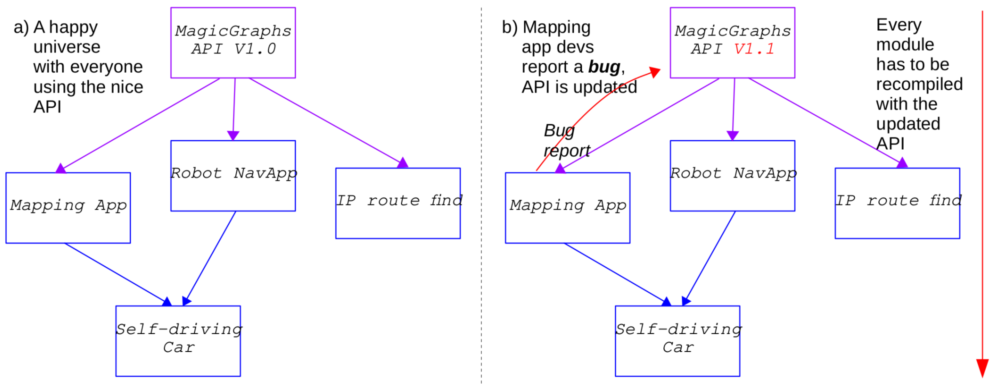

a) A happy universe with everyone using the nice API
MagicGraphs API V1.0
Robot NavApp
Mapping App
IP route find
Self-driving Car
b) Mapping app devs report a bug, API is updated
MagicGraphs API V1.1
Bug report
Every module has to be recompiled with the updated API
Robot NavApp
Mapping App
IP route find
Self-driving Car

Figure 6.3: Updates to the **module** that do not change the function declarations in the **API** still require that all programs using the **module** be recompiled.

This is not uncommon, and should highlight the importance of having a **solid development and testing process** for any **module** that is going to be made available for use by others - the more users out there for a particular **API**, the larger the impact of any **bugs** or **security vulnerabilities**. This will place a burden on the developers of a successful **API** as they will be expected to stay on top of any **bug reports** or **reported security issues**.

A final issue of note is that once the **API** has been released and it's been picked up by a population of developers, it becomes **very difficult to make changes to it** even if we realize that **it was not designed properly** in the first place. To see why this is the case, consider the following situation:

Example 6.2 A developer who wants to use the path finding **module** requests that the path finding functions support **graphs with weighted edges**. These are very common in all kinds of applications, so it is expected that a potentially large number of people who need a module that does path finding would find this useful.

Unfortunately, this is not something we can easily change without significantly impacting any current users of the **API**. The original functions that perform path finding specify that the edge information is **integer** and only **the presence** or **absence** of an **edge** is indicated.

Weighted edges could have any **real value**, so the **integer** data type will not work in general. To provide the desired functionality, we would have to **change the declaration for findPath()** so that it receives an **adjacency matrix** that stores **float** or **double** values (we would also need to change **findPath_l()** to use an **adjacency list** that stores the **edge weights**).

In terms of updating our **module** and **API** this may not seem too big a change - but it has major implications to any existing users of the **API** all of whom have programs that interact with our original **API**. Because we can not simply **typecast** an **integer array** into a **floating point array**, each of the existing users of our **API** will need to change their program so that their graphs (currently not weighted) are stored in a suitable **floating point array** or **adjacency list**. This is shown in Fig. 6.4.

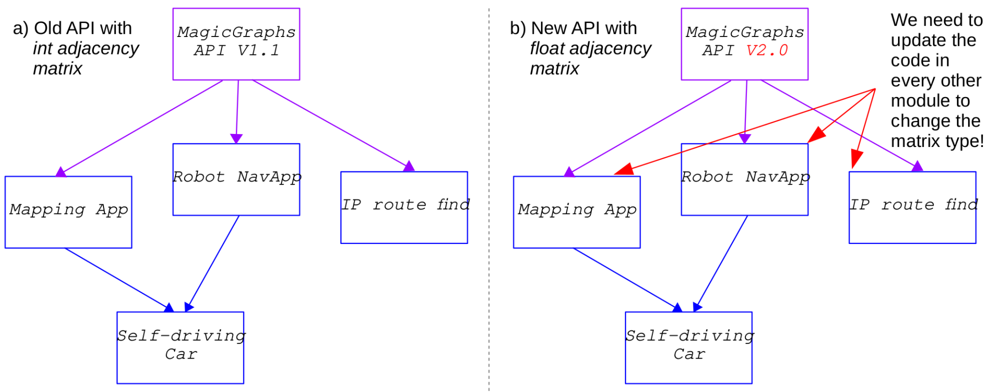


Figure 6.4: Updates to the **module** that **change the function declarations** in the **API** will require all existing users to **modify their own programs** so that they work with the updated **API**. This may involve significant effort, and even if it does not, it is still unwelcome and annoying to developers.

The situation is far from ideal. We have to choose between **making the API more useful to a wider range of developers**, or **avoiding annoyance for existing users**. Note that if we had **thought more carefully** about how to handle **graphs** before writing and releasing our **API** this problem could have been avoided.

We could try to solve the problem by adding **yet another** variant to the **findPath()** function, maybe something called **findPath_d()** which takes a **double precision floating point adjacency matrix**. We may as well go ahead and add **findPath_dl()** which is the equivalent for **adjacency lists**. But now we have **four variants** of the **findPath()** function, with the inherent problem of **code duplication** and the corresponding increase to the work needed to fix **bugs** and keep the **module** up to date.

One last attempt at making everyone happy may involve **simply telling developers how to modify the API themselves** - for instance, by providing them with instructions on how to **patch** their own copy of the **module** so that it supports **weighted graphs**. However, this has two **major drawbacks**:

- Developers using a modified version of the **API** and **module** would be **unable to get updates** unless they are willing to re-do the work of **patching** each update themselves (and this may not even be feasible depending on what was updated).
- If a module using a modified version of the **API** is part of a larger software project that uses other modules that incorporate the **original API**, the larger project **may no longer compile** because of incompatibilities between the versions of the **module** (this is illustrated in Fig. 6.5). There are ways to get around this problem, but they are **band-aid solutions** to a problem that should not exist in the first place, and may not work depending on how different the versions of the **module** become.

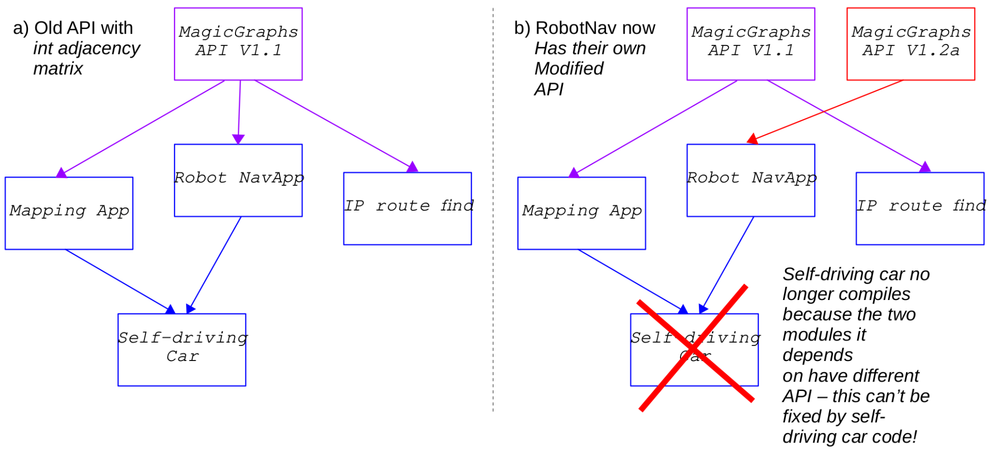

a) Old API with int adjacency matrix
MagicGraphs API V1.1
Robot NavApp
Mapping App
IP route find
Self-driving Car
b) RobotNav now Has their own Modified API
MagicGraphs API V1.1
MagicGraphs API V1.2a
Robot NavApp
Mapping App
IP route find
Self-driving Car
Self-driving car no longer compiles because the two modules it depends on have different API – this can't be fixed by self-driving car code!

Figure 6.5: Allowing developers to **modify the API themselves** could easily cause larger software projects to break, as incompatibilities between versions of the **API** used by different modules would show up during **compilation** and/or **linking**.

All of the above is to motivate the idea that **we have to be very careful** when we set out to design an **API**, and we have to think very hard about what **use cases** the **module** we are developing may be applied to, and about how to write the **API** in such a way that it will be reasonably **easy to maintain and expand** without a significant impact to applications that use it.

Note

The discussion above introduces a couple very important concepts:

- **Use case:** This is simply a **specific situation** or **problem** that a module or software component may be required to handle, or provide support for.
- **Dependency:** It is a **relation** between software **modules** where a particular piece of software **uses** functionality from, and therefore **requires** that a **library or module** be **available** and have the **correct version** in order work.

In **C**, anything we add to our code via **#include** statements introduces a dependency. For instance, our programs almost always depend on the system libraries **stdio**, and **stdlib** at the very least. If these are not present, we can't compile the program. If they have the wrong version, the executable may not run and may have to be re-compiled with the correct version.

6.4.2 Designing a good API

Advice on how to design a good **API** is best received from those who have the most experience designing some of the most widely used **APIs** currently available. The suggestions and advice below are a summary of the advice provided by **Google** in their article **How to Design a Good API and Why it Matters** which can be found here: `https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/32713.pdf`.

According to Google's experts, a good API should be:

- Easy to learn and use
- Difficult to use incorrectly
- Easy to maintain (the code in it is readable and well written)
- Easy to extend and improve
- Suitable for those who will be using it

The principles they propose for us to consider are:

- The **API** should **do one thing, and do it well** - the functionality should be easy to explain and make sense of. If it can't be explained in simple terms, it's probably not well done.
- The **API** should be as **small as possible**, but not smaller. It should satisfy the need it's serving, but should not try to add every single possible thing we can think of. The key idea here is that we can add to it later, but once it's there, it's hard to remove functionality.
- The **API** should be **implementation independent**. This is a particularly important point. It should be possible for us to completely change the way a function is implemented, without needing to change the **API**. This also makes it possible to provide **API** implementations for different systems and platforms - they should be identical as far as the user is concerned.

- Maximize **information hiding** - The **API** should only make available to the user the functionality and data that the user needs. This is harder to do with **C**. We will see shortly how to do a much better job of controlling what data and functions a user of an **API** has access to.
- Names should be **self-explanatory**. Use naming consistently, the same goes for the use of underscores and capitalization.
- **Good documentation** must be provided. Every component of the **API** must be documented properly.
- Think about **performance**, and avoid decisions in the API that will have a negative impact on performance.
- The user of the **API** should **not be surprised** by the behaviour of the **API**.
- The **API** should **report errors as soon as possible** after they occur. It should be clear what the error is, and where it happened.
- If the **API** makes information available in strings, provide functions to parse the string into any components the user may need to handle separately. This prevents annoyance and keeps the user from having to write parsers for the **API**'s output.

There are also a couple of suggestions about good programming practice that are not specific to the design of good **APIs**.

- Use the most **appropriate return value** for each function.
- Where functions have similar lists of parameters, be **consistent with the ordering** of the parameters (reduces the chance of the user making a mistake because they got used to the parameter ordering in a different function).
- Avoid long parameter lists.

The Google document has several more recommendations specific to Java and object oriented code, so make sure to revisit the advice there as you become familiar with object oriented programming. For now, keep in mind the above, and note that even Google developers admit that designing and implementing a good **API** is a hard task, and that we can never achieve perfection.

6.5 Limitations of our programming language

Once we start working with **APIs**, and thinking in terms of **modules** that are **self-contained** and provide functionality to the users without them needing to know the internal details of how this functionality is implemented, we will realize that there are limitations to what we can do with **C** as our programming language.

To illustrate the key issue that should concern us here, suppose we are providing a **module** for creating and managing linked lists of strings (this could be used by an app, for instance, to keep the names of all the eBooks a user has in their cellphone). We have determined the appropriate **API**, and written both our (very well documented) header file, and the corresponding implementation file. Any user needing a linked list can include our module in their code and have all the functionality of linked lists at their disposal. However, **there is a catch** - any programs that use a linked list provided by our **module** must have access to the **head of the list**, so the pointer to the head of the list has to be declared and is owned by **code outside of the module**. The situation is illustrated in Fig.6.6.

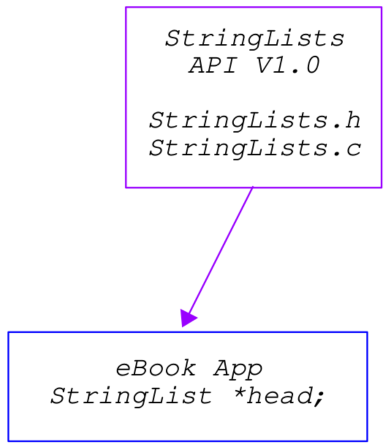


Figure 6.6: A **module** that provides functions to handle **linked lists of strings**. Unfortunately, **the head pointer** has to be declared and is owned by **code that is outside the module**.

This is not great - the user of our **module** has access to all the **CDT definitions** from the **module** (which it needs in order to be able to declare and use the head pointer), and could easily create its own **nodes, pointers, and even lists** without using the **API** functionality. Worse than that, the code outside the **module** has access to all the information stored in the **linked list**, and could easily access, modify, or even delete any of the data stored there without going through **API**. This is a problem because it allows users whether by mistake, or intentionally, to do things with information that the **module** should be handling that were not intended and are not provided by the **API**.

The situation above breaks the concept of a **module being a self-contained entity** that can be used without knowledge of the details of how the module is implemented. It introduces the potential for **bugs** created by users unintentionally modifying data needed by the module, and it introduces **security and privacy** concerns because there is no way to **hide** or **protect** information that should not be available outside of the **module**.

The root cause of this problem is that **C** provides no mechanism for **protecting** sensitive data from being misused or accessed in a way that was not intended by us when we developed the **module**. We can't solve this in **C** without severely impacting the usefulness of our **module**. So at this point we have to look beyond the language we've been using and find a different way to organize our code and data so that we can implement software **modules** that are truly **self-contained**. We have to be able to **control access** to the data managed by the **module** in a way that prevents users from unintentionally, or intentionally, accessing anything we did not intend for them to have access to. This is called **information hiding**, and is one of the fundamental principles of good software design.

6.6 Object Oriented Programming (OOP)

Object Oriented Programming is built around two principles: **Encapsulation**, and **information hiding**.

Definition 6.1 (Encapsulation)

Encapsulation means **wrapping together** *all the components required to implement the functionality of particular* **software module**. *This includes* **all the data** *as well as* **the functions** *that manipulate it.* **Object oriented** *programming languages must provide support for* **encapsulation** *as a central feature of* *the language (as opposed to something that can be achieved by being creative with language features not designed to provide encapsulation).* ♣

Definition 6.2 (Information hiding)

This refers to the ability of the designer of a **software module** *to decide* **which functionality and data** *should be* **visible** *to a user of the* **module***, and to* **hide** *everything else. This includes* **implementation details***,* **data that is essential for the correct working of the module***, and* **functionality** *that is part of how the* **module** *gets its work done but should not be directly accessed by a user.* **Object oriented** *programming languages provide support for* **information hiding** *by allowing a designer to* **control access** *to the* **data** *and* **functions** *that are part of a* **module***.* ♣

To support **encapsulation** and **information hiding**, **Object Oriented Programming** introduces the concept of an **object** as the fundamental unit of **functionality**, which includes **data storage**, as well as **information processing**. We will now spend a bit of time understanding how **objects** are built, what **features they have**, and what we can do with them as software designers. But in order to actually understand what an **object** is, we first need to learn how to design and implement the principles of **OOP** that we discussed above.

6.6.1 Classes

In **Object Oriented Programming**, we expand on the idea of **compound data types** that we discussed in Chapter 3 so that we are not limited to storing data. With **OOP**, we want to **bundle together** all the **data** and all the **functionality** required to manipulate it. The resulting entity is called a **class**. This is illustrated in the figure 6.7.

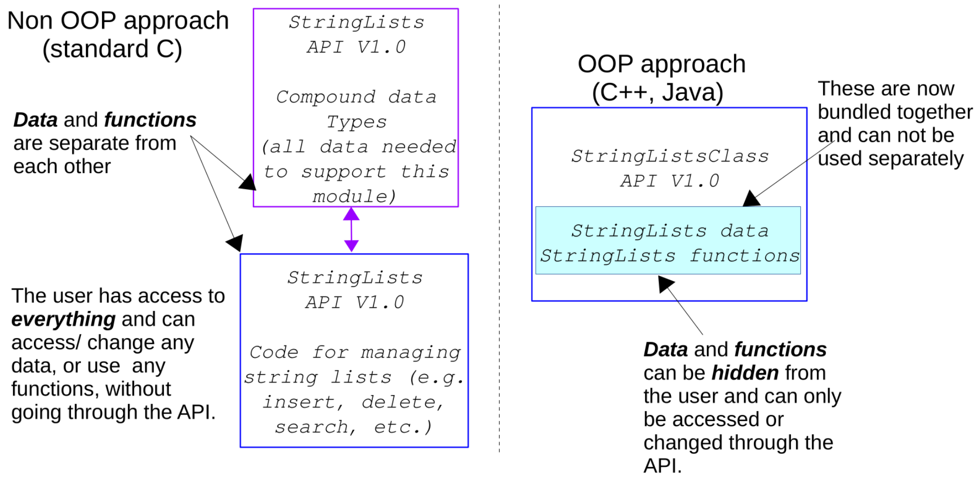

Non OOP approach (standard C)
StringLists API V1.0
Compound data Types (all data needed to support this module)
Data and functions are separate from each other
The user has access to everything and can access/ change any data, or use any functions, without going through the API.
StringLists API V1.0
Code for managing string lists (e.g. insert, delete, search, etc.)
OOP approach (C++, Java)
These are now bundled together and can not be used separately
StringListsClass API V1.0
StringLists data
StringLists functions
Data and functions can be hidden from the user and can only be accessed or changed through the API.

Figure 6.7: A comparison between **non-object oriented** model (a **CDT** plus **separate implementation**) versus an **object oriented** model. In the **OOP** model, everything is **bundled** into a single **class**.

The idea of **bundling together the code and data** that comprise a single module, data type, or data structure, is important; but you have already worked with **CDTs**, so you know there really is nothing fundamentally new in the concept of a **class**. The importance of the **class** idea lies in how the programming language uses classes to provide you with **information hiding** as well as a number of other powerful features that are very difficult (or even impossible) to implement without the support of **object oriented** languages, and that allow us to build software that is conceptually easier to understand, maintain, expand, test, and debug.

In **OOP**, the **data components** of the class are called **member variables**, and the **functions** that provide the functionality for the **class** are called **class methods**.

6.6.2 Information hiding in classes

Recall that in **C** all our **variables** and **functions** have an associated **data type** that is fixed and used by the compiler to determine what code needs to be generated to implement the functionality specified in your program.

In **Object Oriented Programming**, in addition to their **data type**, both the **member variables** and the **class methods** have an associated **access control modifier**. These specify the **visibility** of each of these components much the same way that scope determines the visibility of variables within the code.

The smallest subset of **access control modifiers** that has to be implemented by an **object-oriented** language is:

- **public** - This modifier states that a **member variable or class method** can be accessed by (is visible to) any code whether that code is part of the **class** or whether it belongs to an **external program** using the **class** to do some work. It is appropriate for all the components of the **class** that the user will need to call in order to **use the class** for its intended purpose.
- **private** - This modifier states that a **member variable or class method** can be accessed by (is visible to) exclusively the code that is part of the **class methods**. No external program can **see**, **access**, or **use** any of the **private data** or **methods**.

There are further modifiers that may or may not be available depending on the language we are using, but the two above constitute the minimum subset that will allow us to implement **information hiding**. The diagram in Fig. 6.8 shows how a **class** implementing our string list **API** could be structured - it shows both the **member variables**, and the **class methods**. Access modifiers indicate which components of the **class** are visible to (can be accessed/used by) the user's program. Any **private** methods are only accessible from within the **class**.

In the example from Fig. 6.8, the **head pointer** is not visible to the user, **it can not be changed from outside the class**. However, functions such as **insert()** which are **class methods** can access and change the value of the **head pointer**. In this way, we expose to the user **only the functionality of the class that we want to provide**, and can do so in a way that **prevents misuse of the class and its member variables and methods**.

There are two **methods** we haven't seen before in the class example from Fig. 6.8: the **constructor** and the **destructor**. These two functions have an important role for the **class**:

- **The constructor** - Is a **method** that is **automatically called** by the when we **create** a new **instance of the class** in a program. The **constructor** has the job of initializing the **member variables** and **any data** that the **class** will need to suitable values. In the example on Fig. 6.8, we could expect it to set the **head pointer** to **NULL**, and the **list_length** to **zero**. For more complex classes, the constructor may do a lot more work.

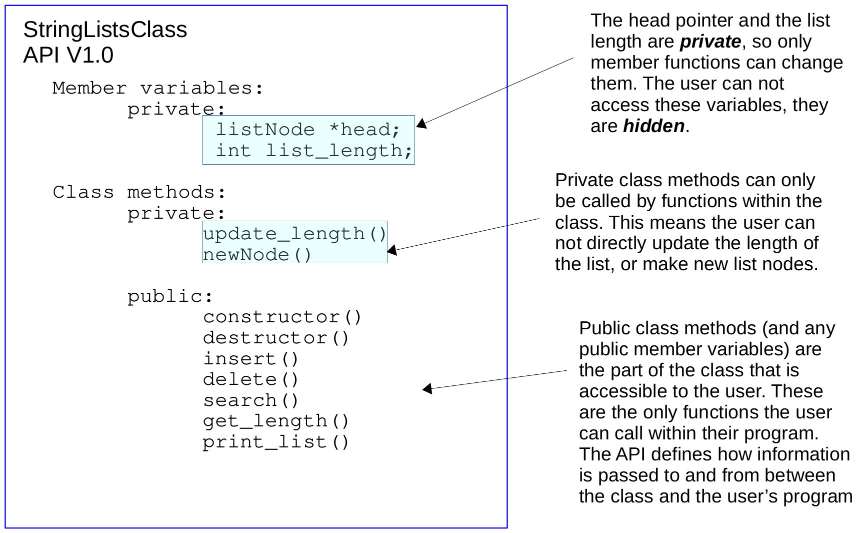

StringListsClass
API V1.0
Member variables:
private:
listNode *head;
int list_length;
Class methods:
private:
update_length()
newNode()
public:
constructor()
destructor()
insert()
delete()
search()
get_length()
print_list()
The head pointer and the list length are ***private***, so only member functions can change them. The user can not access these variables, they are ***hidden***.
Private class methods can only be called by functions within the class. This means the user can not directly update the length of the list, or make new list nodes.
Public class methods (and any public member variables) are the part of the class that is accessible to the user. These are the only functions the user can call within their program. The API defines how information is passed to and from between the class and the user's program

Figure 6.8: Example of **class** structure showing components with different **visibility**. Any part of the class declared as **private** will only be **accessible** from within the **class** itself. Any **public variables and methods** are **accessible** from any code where the **class** is being used.

- **The destructor** - Is a **method** that is **automatically called** when an **instance of the class** goes **out of scope** or when we want to **delete** an **instance of the class**. It has the job of **cleaning up** after the **class**. For instance, in the example from Fig. 6.8, the **destructor** would be in charge or **freeing all memory** allocated to the nodes of the linked list. For more complex classes, the destructor may do a lot more work.

The important thing to keep in mind is that these methods are provided in order to **automate** work so the user of the **class** doesn't have to do it themselves.

Note

Just what is an object? In **non-object oriented** programming, we create **variables** out of any **CDT**. We simply call them **variables** and think of them as **units of data**. The equivalent in **OOP** is a **class instance** or for short **an object**. This is where the **object** part of **object oriented programming** comes from.

The **class** is the template for building **objects**, just like the **declaration of a CDT** is a template for declaring **variables** of that particular **compound data type**.
Each **object** should be thought of as a **big box in memory** that has all the **member variables** and **class methods** specified by the **class declaration**. **Objects** are the essential **functional unit** in **OOP**. The difference between a **class** and an **object** is illustrated in Fig. 6.9.

6.7 Implementing a class in C++

In order to explore some of the features of **OOP** that help us design and implement good software, we need to see how these features work in the context of an actual **Object Oriented** programming language. For this chapter

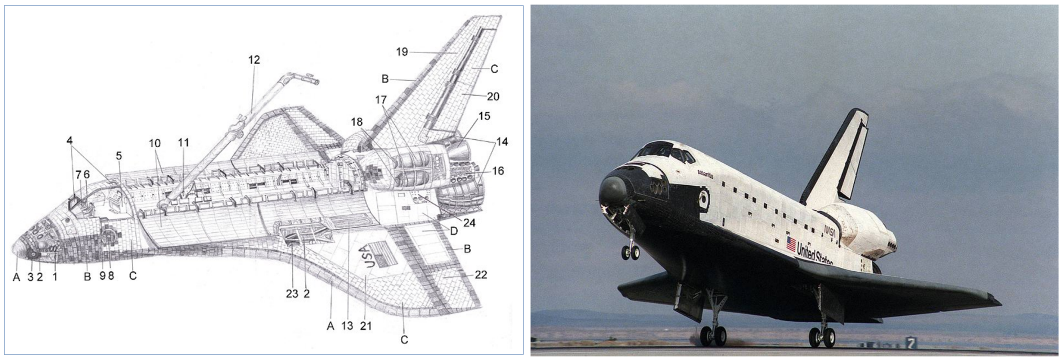


Figure 6.9: The **class** is a blueprint for building **objects**, the **class** specifies all the parts, components, and functionality each **object** must have. From the class, we create **objects** which are specific **instances** of a **class**. *Images: Left-side, Wikimedia Commons, by Eryn Blaireova, CC-SA2.5. Right-side, U.S. National Archives, Public Domain.*

we will use **C++**. This will allow us to apply all the work we have done thus far in learning **C** as we work on developing an understanding of how **object oriented programming** allows us to build software in ways that were not possible with regular **C**.

C++ was developed in the 80's as a significant improvement over standard **C**. It was provided with the syntax and features required to support **object oriented programming**, and has been continuously expanded and improved over the years. **C++** is one of the more important languages in software development. Areas such as operating systems, security, networking, embedded systems, media-related software (e.g. encoding/decoding video and audio), and compilers; among others make extensive use of software written in **C++**. Because it is developed from **C**, the basic building blocks of **C++** programs are already well known to us. Function declarations, variables and standard data types, loops and program control structures are all the same as in **C**. All the regular **C** programs we have written thus far are also valid **C++** programs. But the range and flexibility of features provided by modern **C++** compilers far exceeds anything regular **C** can do.

With that in mind, let's see how a **class** is declared in **C++**, and consider the similarities a **class** declaration has with regard to **compound data types** in **C**. The following listing could be stored in a **header** file called **StringList.h**

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

// Notice the #include statements above. We are using standard
// C libraries (which is allowed since C++ compiles any standard
// C code} for familiarity.
// However, if you want to write pure C++, you should use the
// object oriented libraries that provide much more advanced
// (and object-based) functionality! We will not do that here
// because this is not meant to be a book on C++ programming.
// Here is a good-ol' CDT to store nodes in a
// linked list - for comparison with the class
// just below. There is nothing fancy in the CDT,
// just a bento box with data in it.

typedef struct ListNodeStruct
{
  char string[1024];
  struct ListNodeStruct *next;
} ListNode;

// Here is something new. A StringList class,
// which bundles together all the variables
// and data needed for the list, as well as
// the functions that implement the functionality
// required of the list of strings.
class StringList
{
  // Member variables
  private:
              ListNode *head;
              int list_length;

  // Class methods
  public:
              StringList()
              {
                head=NULL;
                list_length=0;
              }
              ~StringList()
              {
                  clear_list();
              }

              void        insert_string(char string[1024]);
              void        delete_string(char string[1024]);
              void        print_strings(void);
              ListNode    search(char string[1024]);
              void        clear_list();
              int         get_length();

    private:
              void  remove_head(void);

};
```

A few things to note in the example above:

- The **typedef** for the **ListNode** is not part of the **class**, but we need it to define the **nodes** of the string linked list we are building.
- The **class declaration** is just like any other **CDT** declaration we have done before in **C**, except we use the keyword **class** so the **C++** compiler knows we are **bundling** data and functions into one nice package.
- All the **member variables** in our **class** are **private**. No functions or code outside of the **class** can access or change these variables. They are **hidden** from user code.
- The function **StringList()** is the **class constructor**. It has **no return value or type**. It gets called automatically when we create **objects** of this **class**, and will initialize the **class' member variables** as needed.
- The function **~StringList()** is the **class destructor**. Like the **constructor**, it has **no return value or type**, and gets called automatically when an object of the class goes out of scope or gets deleted. Its job is to **clean up** after the class - so in effect, **free any memory** that was dynamically allocated by **class methods**, close any open files, etc.

- The remaining **public methods** are analogous to the functions you would normally find in any regular linked list. We have **insert_string()**, **delete_string()**, and **search()**.
- There is one **private class method**: **update_length()**. This **method** can only be called by other methods within the **class**. It can not be directly called by the user.

In order for the class to be useful, we need to provide an implementation for the **class methods**. The implementation typically goes into a separate program file with the extension **.cpp**. Here is what the implementation looks like for the short class we declared above:

```
#include "StringList.h"

// Notice the 'StringList::' prefix for function declarations.
// This is a class name qualifier and tells the compiler we are
// providing an implementation for a function of the 'StringList'
// class. Without it, the compiler would assume we are
// implementing a regular (non-class member) function that
// happens to be called 'insert_string()'.
// All functions that are part of the class must have the
// corresponding qualifier.

void StringList::insert_string(char string[1024])
{
    // Insert a new string into the linked list
    // at the head of the list (for simplicity)

    ListNode *p=new ListNode;    // This is how we dynamically
                                 // allocate data on-demand in C++
                                 // (no need for calloc)

    if (p==NULL)
    {
        printf("There is no memory for more strings!\n");
        return;
    }

    strcpy(&p->string[0],string);

    if (this->head==NULL)        // 'this' is a pointer to
    {                            // the specific object for
        this->head=p;            // which the function was called
        this->list_length++;
        return;
    }
    else
    {
        p->next=this->head;
        this->head=p;
        this->list_length++;
        return;
    }
}

void   StringList::delete_string(char string[1024])
{
        // Deletes a string from the linked list if it
        // is found there
        ListNode *p, *q;

        p=this->head;
        // If requested string is at the head of the list
        if (strcmp(p->string,string)==0)
        {
            this->head=this->head->next;
            this->list_length--;
            delete p;
            return;
        }

        // Otherwise
        while(p->next!=NULL)
        {
            q=p->next;
            if (strcmp(q->string,string)==0)
            {
                p->next=q->next;
                delete q;
                this->list_length--;
            }
            p=q;
        }
}

void StringList::print_strings(void)
{
    // Print out all the strings currently in the list

    ListNode *p;

    printf("*** The strings currently in our list are:\n");
    p=this->head;
    while(p!=NULL)
    {
        printf("%s\n",p->string);
        p=p->next;
    }
}

ListNode StringList::search(char string[1024])
{
    // Find a node in the list that contains the requested
    // string. Return a *COPY* of the node that has no
    // pointers to list data. If no matching string is found,
    // it returns an <empty> ListNode.

    ListNode *p, copy;

    strcpy(&copy.string[0],"");
    copy.next=NULL;

    p=head;
    while (p!=NULL)
    {
        if (strcmp(string,p->string)==0)
        {
            copy=*(p);
            copy.next=NULL;
            return copy;
        }
        p=p->next;
    }

    return copy;
}

void   StringList::clear_list(void)
{
    // Free all memory allocated to the linked list, reset
    // length to zero, and set head pointer to NULL
    while (this->head!=NULL)
      remove_head();
}

int    StringList::get_length(void)
{
    // This is what in object-oriented programming is called
    // a 'getter'. A function that returns the *value* of
    // a private variable so the user can find out what it is
    // without actually getting direct access to read or change
    // the variable.

    return this->list_length;
}

void   StringList::remove_head(void)
{
    // Removes the node at the head of the linked list
    // and releases the memory allocated to the list
    // node. It reduces the length of the list by 1
    //
    // This function is used by the function that
    // clears the linked list. We do not give the users
    // of the class access to it because they could
    // use it to remove list data without
    // regard for the contents. Hence, we make this
    // function private.

    ListNode *p;

    if (this->head==NULL) return;

    p=this->head;
    this->head=this->head->next;
    delete p;
    this->list_length--;
}
```

A few **syntax** notes are worth a quick look:

- The declarations for the **class methods** have to be preceded by the name of the **class** and a '**::**'. This is used by the compiler to figure out which **class** each function is a part of (there may be many different **classes** with functions that have the same name, or there can be functions that are not part of any **class** that have the same name as a **class member**.
- We no longer have to use **malloc()** or **calloc()** to **dynamically allocate memory**. **C++** provides a keyword

new that allocates and initializes new memory on request, as well as a corresponding **delete** keyword that releases space once we are done using it. These two keywords exist because with objects some extra work has to be done. Whenever we create a new **dynamic object**, the **constructor** has to be called. The **new** operator does this. Similarly when we are done using an object and want to release its memory, the **destructor** has to be called. The **delete** operator takes care of that.

- To access any of the **variables or class methods** from **within the class member functions**, we use the **'this'** pointer. It works just like any other pointer in **C** and **C++** and we can get to any **component of the class** by using the **-> (arrow)** operator. What's up with **'this'**? If we think about it, to access a **field** in a **CDT** we use **variable_name.field_name** (or **pointer_name->field_name**). However, **we do not know the name the object will have in advance**. The user will call their **objects** whatever they want, and there will most likely be many **objects** with different names from the same **class** - so we can't use the **object's name** while writing the implementation of the **class member functions**. The **'this'** pointer is a convenient way for the compiler to know that whatever name the user gives an **object**, a **member function** needs access to a specific **field** within that particular **object**.

At this point, it's worth spending a moment or two thinking about how the **StringList class** we just created compares to the **non-object oriented** linked lists we developed in Chapter 3. The fact that both the **data** and **functionality** are **bundled** together as part of the **StringList class** makes the design of the software **cleaner**, **more intuitive**, and **easier to understand**. Much like **CDTs** allowed us to represent complex data items as **individual units of information**, **classes** allow us to represent software components as **individual units of functionality**.

Not only that, with our original linked lists written in **C** there was no protection against a user of the code accidentally (or maliciously) accessing sensitive information (such as, for instance, the pointers that keep the list organized and properly linked). The **StringList class** uses **information hiding** to ensure that users have no access to such sensitive **variables** and/or **class methods**. To understand how this works, let's look at a very simple program that creates **one object** of the **StringList class**, and then tries to access **private variables** or **private class methods**.

```
#include "StringList.h"

int main(void)
{
   // This is a very short example of using the StringList library to
   // create a linked-list of strings.

   StringList my_list;       // Here is an actual object of the
                             // StringList class.

   // Let's see what happens if we try to access private variables or
   // methods:

   printf("The length of the list is: %d\n",my_list.list_length);

   // Try and call a private class method
   my_list.remove_head();

   return 0;
}
```

If we try to compile the program above, we will see the following:

```
> g++ -c stringExample.cpp
stringExample.cpp: In function 'int main()':
stringExample.cpp:14:54: error: 'int StringList::list_length' is private within this context
    printf("The length of the list is: %d\n",my_list.list_length);
                                                     ^~~~~~~~~~~
In file included from stringExample.cpp:1:0:
StringList.h:33:21: note: declared private here
               int list_length;
                   ^~~~~~~~~~~
stringExample.cpp:17:25: error: 'void StringList::remove_head()' is private within this context
    my_list.remove_head();
                      ^
In file included from stringExample.cpp:1:0:
StringList.h:55:23: note: declared private here
               void remove_head(void);
                     ^~~~~~~~~~~
```

The compiler will throw an **error** any time that user programs attempt to access **private data or methods** contained in an object. There is **no way** to build a working executable program unless these errors are resolved by removing any attempts to access **private** components of the **class**. In this way, the compiler enforces our decision to **hide certain parts of our class** from users of the **module**.

Let's now see a short example of a program using our **StringList class** as intended, to store and manipulate strings:

```
#include "StringList.h"

int main(void)
{
   // This is a very short example of using the StringList library to
   // create a linked-list of strings.

   StringList my_list;      // This is is an actual object (and instance)
                            // of the StringList class.

   printf("Adding a couple of strings to the list...\n");
   my_list.insert_string("First String");
   my_list.insert_string("Second String");
   printf("The length of the list is %d\n",my_list.get_length());
   my_list.print_strings();

   printf("Add one string and delete another...\n");
   my_list.insert_string("Third String");
   my_list.delete_string("First String");
   printf("The length of the list is %d\n",my_list.get_length());
   my_list.print_strings();

   printf("Clear the list of strings...\n");
   my_list.clear_list();
   printf("The length of the list is %d\n",my_list.get_length());
   my_list.print_strings();

   return 0;
}
```

The listing above should not be too surprising. But it illustrates just how **clean**, **intuitive**, and **easy to follow** a program becomes when it uses **objects** to do its work. The equivalent **C** code would have much longer and more cumbersome function calls (which require additional parameters) and would be **longer** and **less tightly integrated** than the **object oriented** program shown above.

As a very simple example, consider the task of figuring out the **length of the list**. In the **C** implementation there was no easy way to keep track of this value, and we had to do a **list traversal** to count the number of entries every time we needed to find the length. In the **StringList class** this is easily resolved by having a **member variable** that keeps track of the length, and that is directly manipulated by functions that insert or remove entries from the list. This removes the need for a **list traversal**.

Compiling and running the code above produces the following output:

```
> ./a.out
Adding a couple of strings to the list...
The length of the list is 2
*** The strings currently in our list are:
Second String
First String
Add one string and delete another...
The length of the list is 2
*** The strings currently in our list are:
Third String
Second String
Clear the list of strings...
The length of the list is 0
*** The strings currently in our list are:
```

6.7.1 Method Overloading

Recall that as we were designing a simple **API** for path finding, we ran into a situation where one some of the users of our **module** wanted to use an **adjacency list** instead of an **adjacency matrix**. In **C** there is no clean solution that allows users to use either of these (as needed) to access the functionality provided by our **module**, and we were stuck creating multiple functions with similar names and **significant amounts of code duplication** in order to provide the needed functionality. Code duplication is not a good thing and should be avoided whenever possible. We will soon see ways in which **object orientation** allows us to elegantly expand and refine an **object's** functionality without unnecessary code duplication. But first, let's look at a feature of **object oriented** languages that allows us to avoid having multiple functions with similar names all of which are intended to provide the same functionality.

In **C++** and other **object oriented** languages, we are allowed to declare multiple functions **with the same name but different arguments and/or return value types**. This is called **method overloading**.

In our **API** example, we had a situation where some users needed a function that worked with **an integer adjacency matrix**, some users wanted a function that worked with a **floating point adjacency matrix** so they could use it on **graphs with weighted edges**, and some users wanted a function that worked with **an adjacency list** (we decided we should also support **weighted** and **non-weighted edges**). This created four functions with different names in **C**. With **C++** the situation is different, we are allowed to declare the functions as follows:

```
intList *findPath(int Adj[N][N], int N, int start, int goals[k], int k);
intList *findPath(double Adj[N][N], int N, int start, int goals[k], int k);
intList *findPath(intList* A[N], int N, int start, int goals[k], int k);
intList *findPath(floatList* A[N], int N, int start, int goals[k], int k);
```

This may not seem like a big change - after all we still have **four functions** with **slightly different functionality** which all do the same thing. However, they all have the **exact same name** which makes it **conceptually easier** for a developer to work with them. **Method overloading** makes the code for programs using a **module** easier to read, and allows an **API** to provide the functionality required in a **clean** and **concise** way. With an **object oriented** language, there is no possible confusion between **functions that have similar names but provide different functionality**, and **variations of the same function which have different names because the language doesn't support overloading**.

How does **method overloading** work in practice? The compiler has access to all the different variations of the function, when a user program calls the function, the compiler **checks the list of arguments passed to the function** and **finds a matching definition** among the **overloaded methods** we provide in our **module**. If a matching method is found, the compiler will use the correct function all. Otherwise it reports that there is no matching method for the specific list of parameters the user provided.

Just a word of caution: While we can provide as many **different overloads** for any given **method** as we want, we should not simply **create code blot** by providing every possible combination of parameters we can think of. Instead, every **version** of an **overloaded method** that we decide to add to the **API** should be supported by a valid **use case**.

Note

Method overloading is an example of what in **object oriented programming** is called **polymorphism**. **Polymorphism** is a term for a thing that has multiple different shapes. In the case of **method overloading**, it refers to a function that has the same purpose (same functionality, same name) but different **lists of parameters and/or return type**. We will soon see that **polymorphism** also occurs at the level of **objects** and that it provides us with incredible power in terms of **building functionality** into our **classes**.

6.8 Inheritance and class hierarchies

A very common situation in software design involves writing programs that can work with a variety of data items that are **variations, or refinements of each other**. For example, software to run a library catalog will need to handle records for **books, magazines, journals, periodicals, graphic novels** and other media types. These are all related, they have some **common properties** (e.g. a name, a date of publication, a library code), and some **different attributes** that are specific to each particular type of item. As another example, consider a **media player** application - it is required to handle a large variety of **media types** and **media formats**. The corresponding **media files** have **common properties** as well as **attributes that are specific to each media type**.

Handling such situations without **object oriented programming** results is code that is **cumbersome**, **hard to maintain, test, debug, and expand**, and that contains **significant amounts of code duplication**. To see why this is the case, consider the following concrete problem:

Example 6.3 We are implementing a **software synthesizer**. This is a program that takes a music file that contains **information about notes in a song**, the **timing and duration of each note**, and also **the instrument that plays**

each note. The **synthesizer** then **generates sound** to play the song. One way to think of it is this: Give the **software synthesizer** a **musical score**, and it will **play the music** shown in that score. It is **not the same as a recording** such as you would get from a streaming service or store on your smartphone - they key is that **the user can change the score** and the music will change accordingly.

A key component of the **software synthesizer** is a **list of notes** that are playing at a particular time. Each of these is then processed to generate the corresponding sounds. In **C**, we would normally represent a note as a **CDT** using something like this:

```
typedef struct note_struct
{
    double time_position;    // Note's position in the musical score
    double frequency;        // The note's frequency
    double duration;         // How long the note is played
    int    volume;           // The loudness of the note
    int    instrument_ID;    // Indicates which instrument this note
                             // should be played with
    double *note_data;       // Pointer to the note's data array
} Note;
```

But if you think about it, this is very **limiting**. Different instruments may require additional information in order to play a note properly. For example, if a note is being played by a guitar, we may need to know which of the guitar's strings is used to play it (which will change the sound). If in addition the note is being played by an electric guitar, we may need to know the state of the **distortion pedal** or the **wah-wah** lever. The point is, **if we are using CDTs**, and we need the software to work with **notes** as a fundamental unit of information, we would need to pack **every single piece of information that any of the instruments may need** into the same **CDT** - there is no way to **modify or adapt** the **CDT** for each instrument. The resulting **CDT** will look like so:

```
typedef struct note_struct
{
    double time_position;    // Note's position in the musical score
    double frequency;        // The note's frequency
    double duration;         // How long the note is played
    int    volume;           // The loudness of the note
    int    instrument_ID;    // Indicates which instrument this note
                             // should be played with
    int    stringNo;         // Which string is used (for guitar only)
    double pedal_val;        // State of the pedal (for electric guitar only)
    double wahwah;           // State of the wah-wah level (electric guitar only)
    double soft_p;           // State of the soft pedal (piano only)
    double damper_p;         // State of the damper pedal (piano only)
    .
    .                        // Many, many more variables needed by each of the
    .                        // different instruments
    .
    double *note_data;       // Pointer to the note's data array
} Note;
```

This is not a very elegant solution, and also wastes a significant amount of storage space, as most notes will use only a handful of the variables contained in the **CDT** to generate their sound.

Having a big and bulky **CDT** is not ideal, but it is not our only problem. Somewhere in the **software synthesizer** code there will be a function whose job is to generate the actual sound data for a particular note. You can imagine a **loop** that goes over each of the notes that need to be played at some point in time, and for each node, it calls the function that actually produces the corresponding sound. The sound producing function will look as shown below:

```
   double get_sound_sample(note *my_note, double time_index)
   {
      // Computes and returns the sound value for the specified note
      // and time index.

      if (note->instrument_ID==0)
      {
         // Do the processing required to get a sound value
         // for instrument 0, using the required variables
         // from the CDT
      }
      else if (note->instrument_ID==1)
      {
         // Do the processing required to get a sound value
         // for instrument 1, using the required variables
         // from the CDT
      }
      else if (note->instrument_ID==2)
      {
         // Do the processing required to get a sound value
         // for instrument 2, using the required variables
         // from the CDT
      }
      else if ...    // for as many instruments as the synthesizer
                     // supports (this can be hundreds!)

 }
```

Within each **if...else** block, there will be code that produces sound for a specific note and specific instrument. A significant portion of this code **will be identical**, so there is a lot of code duplication going on here. Beyond being **long, cumbersome, and hard to read** for a developer, we have a serious problem in terms of **maintaining, expanding, testing, and debugging** this function. With so much duplicated code, if we find a **bug** chances are we will have to apply fixes (and then test them) at multiple places in this function. This increases the likelihood we will miss something. Expanding the functionality of the function (e.g. to add another instrument) only makes this situation worse.

In a situation like this, it is not difficult for the code to become **unmanageable**, **unmaintainable**, and eventually **stale** as the effort required to keep it up to date, fix bugs, and add functionality becomes too great.

We would like to solve a problem similar to the one we just described above, but at the same time we need to:

- **Maximize code reuse** - Any code that is shared among the different variations of our items should be implemented only once.
- **Minimize data duplication** - Common variables and data should be declared only once.
- We want to be able to **extend and/or refine** the behavior of any of our items without affecting the rest.
- We want our resulting software to be organized in a way that makes the relationship between our data items clear, and easy to understand conceptually.

6.8.1 Hierarchies of objects

With **object orientation**, we can build programs that take advantage of the idea that **related objects** can be organized into **hierarchies** in such a way that **shared characteristics and functionality are not duplicated** but we can, at the same time, **provide any level of refinement** that we need for each of the **different objects** in the **hierarchy**.

For our **software synthesizer** and the various types of **notes** it has to handle, a possible hierarchy would look like the diagram shown in Fig. 6.10.

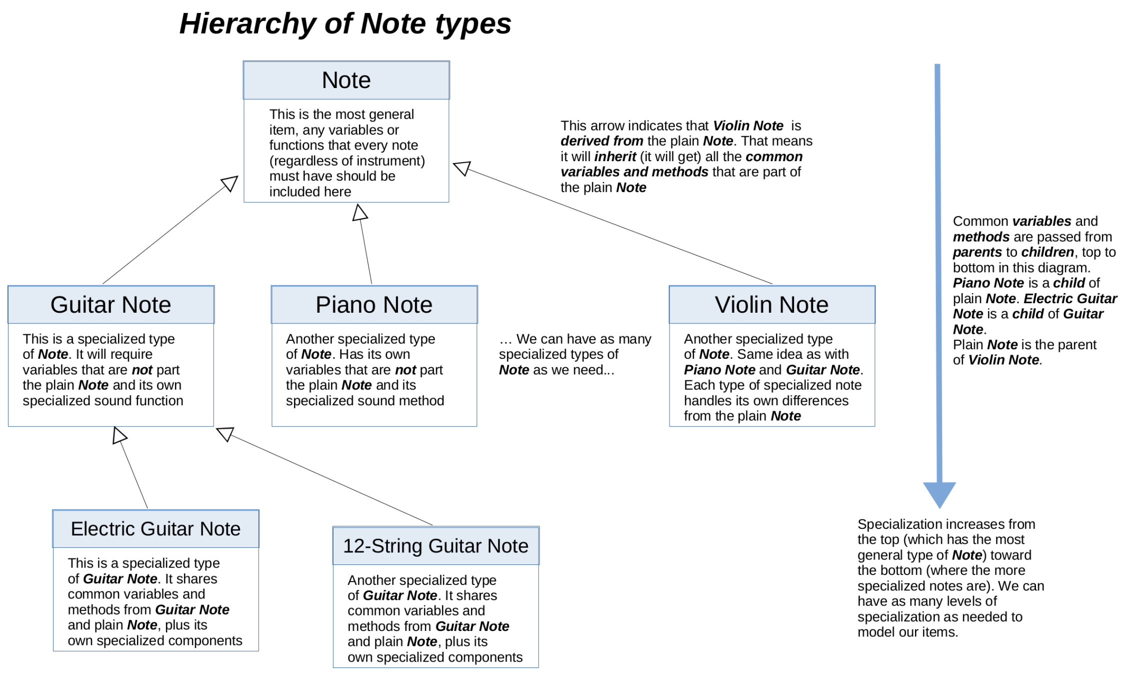

Hierarchy of Note types
Note
This is the most general item, any variables or functions that every note (regardless of instrument) must have should be included here
This arrow indicates that Violin Note is derived from the plain Note. That means it will inherit (it will get) all the common variables and methods that are part of the plain Note
Common variables and methods are passed from parents to children, top to bottom in this diagram. Piano Note is a child of plain Note. Electric Guitar Note is a child of Guitar Note. Plain Note is the parent of Violin Note.
Guitar Note
This is a specialized type of Note. It will require variables that are not part the plain Note and its own specialized sound function
Piano Note
Another specialized type of Note. Has its own variables that are not part the plain Note and its specialized sound method
... We can have as many specialized types of Note as we need...
Violin Note
Another specialized type of Note. Same idea as with Piano Note and Guitar Note. Each type of specialized note handles its own differences from the plain Note
Electric Guitar Note
This is a specialized type of Guitar Note. It shares common variables and methods from Guitar Note and plain Note, plus its own specialized components
12-String Guitar Note
Another specialized type of Guitar Note. It shares common variables and methods from Guitar Note and plain Note, plus its own specialized components
Specialization increases from the top (which has the most general type of Note) toward the bottom (where the more specialized notes are). We can have as many levels of specialization as needed to model our items.

Figure 6.10: A diagram illustrating the hierarchical relationships between different types of **notes**. The idea being that we can build functionality by starting with more general (abstract) items, and then creating multiple **specialized** versions of them that share **common attributes**.

The diagram illustrates the notion that there are **attributes** (**variables and functionality**) that all **notes** must have. It makes sense to bundle them together into a plain **Note**. We can then **derive** different versions of the plain **Note**, one for each of the instruments our synthesizer supports. The **specialized** note types will **inherit** all the **common attributes** of their parent (the plain **Note**), and add their own **specialized attributes** that are required to generate their particular type of sound. We can create as many levels of specialization as we need, so for example we can **derive** various specialized types of **Guitar Note**, or **Piano Note**, or **Violin Note** in order to further enrich our synthesizer's capabilities.

Because our different **notes** are organized in this hierarchical fashion, there is **little or no duplication** of either **data** or **functionality**. Each **specialized** type will define **only** those **attributes** that are **not shared** with other types of **note**, and that make each specialized type unique. Any **common attributes** are created once and then shared according to the relationships described by the hierarchy.

6.8.2 Implementing hierarchies and inheritance

Object oriented languages allow us to implement hierarchies like the one shown in Fig.6.10 by allowing us take any **class**, and **derive specialized versions** of it. The **derived classes** are often called **child classes**, while the original **class** is usually called either **base class** or **parent class**. Let's see how that would work in the example of the **software synthesizer** and see how the use of **class hierarchies** results in a program that is much **cleaner**, and easier to **read, maintain, test, and debug**.

Example 6.4

One of the first step in designing the **software synthesizer** would have been to **figure out the hierarchy of object types** that we need to handle. This is not always straightforward and should receive careful attention because once the hierarchy is implemented, it can be difficult to change it in a significant way. So as part of our design process (as described in Chapter 2), with **OOP** we have to spend a good amount of time figuring out how to organize our data and functionality into **classes** and **class hierarchies**, and we need to figure out **which attributes will belong in which class** within the hierarchy. This work is well worth the effort, as a well designed **class hierarchy** will make our work all that much easier when implementing, testing, and debugging our software.

For this example, we will work with the hierarchy shown in Fig. 6.10 and implement the plain **Note** as well as **two child classes**, the **Guitar Note** and the **Piano Note**. This is simply to illustrate how the process works and the principles involved in building and using a simple class hierarchy. The example can easily be extended to incorporate more **child classes** at different levels of specialization.

Let's have a look at the implementation of the **Note class** which is the **parent class** for both **Piano Note** and **Guitar Note**.

```
class Note
{
 // Member variables
 protected:
             double time_position;   // When the note starts
             double frequency;       // The note's frequency
             double duration;        // Duration in milliseconds
             int volume;             // Volume in 0-10
             double *note_data;      // Data array for the note

 // Class methods
 public:
             Note()
             {
               // Default constructor, used when we do something
               // like this: my_note = new Note;
               this->time_position=0;
               this->frequency=0;
               this->duration=0;
               this->volume=0;
               this->note_data=NULL;
             }

             Note(double t, double f, double d, int v)
             {
               // A constructor that initializes the various values
               // for this note, used when we do something like
```

```
                  // this: my_note = new Note(1.25,440,1.5,7);

                  // It is an example of member overloading. The
                  // compiler chooses the right constructor based
                  // on the parameters we specify while creating the
                  // Note!

                  this->time_position=t;
                  this->frequency=f;
                  this->duration=d;
                  this->volume=v;
                  this->note_data=NULL;
                }

                virtual ~Note()
                {
                  printf("* Note class destructor called!\n");
                  // Release memory for this note!
                  if (note_data!=NULL) delete note_data;
                }

                virtual double get_sound_sample(double time_idx)
                {
                    // This function would generate the
                    // correct sound value for a plain Note
                    // (maybe sounds like a beep?) at the
                    // specified time index, given the
                    // note's frequency, duration, and volume.

                    // Here, for simplicity, it will simply
                    // print a message

                    printf("A plain Note with frequency %f is making sound at time %f\n",this->
                        frequency,time_idx);
                    return 0;
                }

                virtual void print_note_info()
                {
                    // Prints out the values of the note's
                    // variables
                    printf("This generic Note has frequency %f\n",this->frequency);
                    printf("This generic Note has duration %f\n",this->duration);
                    printf("This generic Note has volume %d\n",this->volume);
                    printf("This generic Note plays at time index %f\n",this->time_position);
                }
  };
```

The **Note class** contains all the **attributes** that are common across **all the possible types of notes** that our synthesizer may need to handle. This includes a few key **member variables**, and **two methods** that are common to all notes. A couple of notes on syntax:

- The **protected** access modifier is something we didn't see before. It allows us to **share common attributes** amongst **objects** that are part of a **class hierarchy** while preserving **information hiding**. Specifically, **protected** components of a class are **visible** within code in the **derived classes** but are **hidden from code that is not part of the parent or derived classes**. As far as code **outside the classes** is concerned, **protected** components behave as if they were **private**. We do not use the **private** access modifier here because any **private** attributes would **not be visible** within **derived classes** (notice that they still can be used by a derived class object through any inherited methods implemented in the base class). Having both **private** and **protected** access modifiers allows us to have very fine control regarding how **class attributes** are **shared or not** across a hierarchy, and at the same time enforce **information hiding**.

- There are **two constructors**, the **default constructor** which takes no parameters and initializes the **class**' attributes to default values, and a **constructor** that accepts a list of initialization values for the **data members** of the **class**. This is an example of **method overloading** and is very common - often we want to provide different **constructors** corresponding to different ways in which users may need to create **objects** of a given **class**.
- The **get_sound_sample() method** is declared as **virtual** - this tells the compiler that we expect this function to be **specialized** by the **derived classes**. This doesn't mean that **child** classes are forced to re-implement the function, it simply means that they **could** and **often will** do just that. If the **derived class** provides their own version of **get_sound_sample()**, then their **specialized** version will be used. Otherwise the implementation from the **Note** class will be used.

In the listing above, we created a very simple *dummy* function to generate sound samples. It doesn't actually make any sound, it prints a message instead - we will use this in a moment to see how the class hierarchy works, but in a real implementation there would be a whole lot of code in this function, related to computing and returning the appropriate sound value for the **Note** at the specified time index.

Now that we have the **parent class**, let's see how we would create the two **child classes**: **Piano Note** and **Guitar Note**:

```
class PianoNote : public Note
{
    // This is a derived or child class. The parent is 'Note'
    // and the 'public' access modifier states that:
    // All 'public' attributes of 'Note' will be 'public' in PianoNote
    // all 'protected' attributes of 'Note' will be 'protected' in PianoNote

    protected:
            double soft_p;    // State of the soft pedal
            double damper_p;  // State of the damper pedal

    public:
            PianoNote():Note()
            {
                // Default constructor - it calls the default constructor of the parent class!
                this->soft_p=0;
                this->damper_p=0;
            }

            PianoNote(double t, double f, double d, int v, double sp, double dp) : Note(t, f, d,
                 v)
            {
                // Constructor with initialization data, calls the corresponding constructor
                // in Note for initializing common attributes.
                this->soft_p=sp;
                this->damper_p=sp;
            }

            ~PianoNote() override
            {
                // Should do anything needed to clean up after
                // PianoNote information *but* not the common
                // information from 'Note', since the destructor
                // for 'Note' will be called immediately after this
                // function is done.
                // So, here, we don't need to do anything at all...
                 printf("** PianoNote class destructor called!\n");
            }

            double get_sound_sample(double time_idx) override
            {
                // This is the implementation of the get_sound_sample
                // which should be different from a PianoNote. Here
                // we will simply place a print statement to indicate
                // this function is being called

                printf("A PianoNote with frequency %f is playing at time index %f\n",this->
                    frequency,time_idx);
                return 0;
            }

            void print_note_info() override
            {
                // Prints information for this PianoNote
                printf("This PianoNote has frequency %f\n",this->frequency);
                printf("This PianoNote has duration %f\n",this->duration);
                printf("This PianoNote has volume %d\n",this->volume);
                printf("This PianoNote's soft pedal is at %f\n",this->soft_p);
                printf("This PianoNote's damper pedal is at %f\n",this->damper_p);
                printf("This PianoNote plays at time index %f\n",this->time_position);
            }

};
```

Consider how easily we created a new type of **note**. We used the **common attributes** from **Note**, and simply added the **data members** that are specific to the **PianoNote**. Just two variables, because **PianoNote** will inherit **frequency, duration, time_position, and volume** from **Note**. We are actually using those **common attributes** within the implementation for **PianoNote**, for instance, in the function that prints the values of the **PianoNote** variables. But we don't have to declare them within the **PianoNote class**, they are passed down from the **parent** class.

Notice as well that the **constructors** for **PianoNote** are short and do not duplicate code - we simply call the constructor for the parent class and let that constructor deal with the **common attributes**. The **PianoNote** constructors need only worry about the attributes that are specific to piano notes. Similarly, the **destructor** for **PianoNote** doesn't need to do anything - because the compiler will call the **Note** destructor automatically after the **PianoNote** destructor has done its work, so as to ensure that proper cleanup of **common attributes** is performed.

Finally, and importantly, notice that **PianoNote** provides **its own implementation of get_sound_sample()**. This is possibly the most important bit since it is what allows **PianoNote** to **refine** or **specialize** its behaviour over that of a generic **Note**. This is called **method overriding** (do not confuse this with **method overloading** which we discussed earlier) and is **explicitly indicated in the declaration of the function**.

When the **get_sound_sample()** function is called, the compiler checks if the **child class** has provided an **override**, and if so, it uses the **child class** function. If, on the other hand, the **child class** did not provide its own implementation, then the **parent's class** version is called.

This provides us with the flexibility to decide which functionality will be **refined** by the **child class**, and which functionality will be identical to what is provided by the **parent class**.

To complete the example, here's the declaration for **GuitarNote**:

```
class GuitarNote : public Note
{
   // GuitarNote is also a child of Note

   protected:
           int  stringNo;     // Which string was plucked

   public:
           GuitarNote():Note()
           {
               // Default constructor - it calls the default constructor of the parent class!
               this->stringNo=0;
           }

           GuitarNote(double t, double f, double d, int v, double strNo) : Note(t, f, d, v)
           {
               // Constructor with initialization data, calls the corresponding constructor
               // in Note for initializing common attributes.
               this->stringNo=strNo;
           }

           ~GuitarNote() override
           {
               // Nothing to do here, the compiler will call the destructor from
               // 'Note' after this function is called to ensure proper cleanup.
               // GuitarNote didn't add any dynamic data that needs cleanup!
               printf("** GuitarNote class destructor called!\n");
           }

           double get_sound_sample(double time_idx) override
           {
               // This is the implementation of the get_sound_sample
               // which should be different from a GuitarNote. Here
               // we will simply place a print statement to indicate
               // this function is being called

               printf("A GuitarNote with frequency %f is playing at time index %f\n",this->
                   frequency,time_idx);
               return 0;
           }

};
```

The definition for **GuitarNote** is similarly small, and doesn't duplicate any of the work that is already done in **Note**. Notice that we did not create a specialized version of **print_note_info()**, so if we call this function from a **GuitarNote**, we will in effect be using the function inherited from the **parent class**.

To complete the example, let's see how, once we have spent time and energy building a nice **class hierarchy** of notes, we can easily write a program that is **easy to read and understand**, **doesn't have to deal with the complexities of different types of notes**, and yet **is able to use any of the types of notes we have provided**.

```
int main()
{
   // Let's declare an array of pointers to 'Note' objects
   // so we can simulate a loop that would 'play' music
   // from a set of notes.

   Note *all_notes[10];

   // Pay close attention, the array is for 'Note' objects!
   // but see what we can do now that we have a class hierarchy:

   all_notes[0]=new Note(1.25, 440, 2.0, 7);           // A generic 'Note'
   all_notes[1]=new PianoNote(1.5, 880, 1.0, 5,0,0);   // A 'PianoNote'
   all_notes[2]=new GuitarNote(2.0, 550, .5, 8,1);     // A 'GuitarNote'

   // We don't need to fill the remaining entries, but we could.
   // The key point here is: The program thinks its working with
   // 'Note' objects - but we can put *any derived* class objects
   // in this array. Why?
   // Because a PianoNote is a Note (just a specialized one!)
   // and a GuitarNote is also a Note
   // So software that works with 'Note' objects can use
   // any of the specialized versions without modification!

   // Let's see what happens when we want to 'play' sound from
   // these notes:

   printf("******************\n");
   printf("Playing all notes:\n");
   all_notes[0]->get_sound_sample(1.0);
   all_notes[1]->get_sound_sample(1.0);
   all_notes[2]->get_sound_sample(1.0);
   printf("\n");

   // Notice that the code above doesn't need to worry about the
   // fact each of these notes is actually a different type of
   // object! but the correct behaviour is obtained. For the
   // PianoNote, the PianoNote function is used, for the GuitarNote
   // the corresponding function is called.

   // Now see what happens when we call the print_note_info()
   // function:

   printf("******************\n");
   printf("Printing info for a generic Note:\n");
   all_notes[0]->print_note_info();
   printf("\nPrinting info for a PianoNote:\n");
   all_notes[1]->print_note_info();
   printf("\nPrinting info for a GuitarNote:\n");
   all_notes[2]->print_note_info();
   printf("\n");

   // The last call above is interesting. GuitarNote does not provide
   // a specialized version of 'print_note_info()', so the
   // function from 'Note' is automatically called instead!

   // That's it for this short example, let's clean up!
   printf("Deleting a generic Note\n");
   delete all_notes[0];
   printf("\nDeleting a PianoNote\n");
   delete all_notes[1];
   printf("\nDeleting a GuitarNote\n");
   delete all_notes[2];

   return 0;
}
```

The most important thing to notice in the listing above is that the program concerns itself only with **Note** objects - it doesn't know or care about **PianoNote**s or **GuitarNote**s (and if we had many, many more derived note classes, it wouldn't worry about those either). The program defines an array of **Note** objects, and then proceeds to use it to handle all of the types of notes that exist in our **class hierarchy**.

This works for a reason that is both elegant and intuitive: a **PianoNote** is a **Note**. A **GuitarNote** is also a **Note**. And if we had defined **ElectricGuitarNote** objects, we would realize that they are **GuitarNote**s and therefore also **Note**s. The result of this is that if we write a program that can handle **Note**s, the same program will be able to handle every derived class that is a specialized version of a **Note**. This is a fundamental idea in **OOP** and makes software design and implementation a lot easier and cleaner. Compare the code above with the long, cumbersome, and repetitive **get_sound_sample()** we tried to implement in **C**, and it should be clear that **OOP** has allowed us to create software that is significantly better in terms of the properties we described earlier in this Chapter.

Compiling and running the program in the listing above results in this output:

```
> ./a.out
******************
Playing all notes:
A plain Note with frequency 440.000000 is making sound at time 1.000000
A PianoNote with frequency 880.000000 is playing at time index 1.000000
A GuitarNote with frequency 550.000000 is playing at time index 1.000000

******************
Printing info for a generic Note:
This generic Note has frequency 440.000000
This generic Note has duration 2.000000
This generic Note has volume 7
This generic Note plays at time index 1.250000

Printing info for a PianoNote:
This PianoNote has frequency 880.000000
This PianoNote has duration 1.000000
This PianoNote has volume 5
This PianoNote's soft pedal is at 0.000000
This PianoNote's damper pedal is at 0.000000
This PianoNote plays at time index 1.500000

Printing info for a GuitarNote:
This generic Note has frequency 550.000000
This generic Note has duration 0.500000
This generic Note has volume 8
This generic Note plays at time index 2.000000

Deleting a generic Note
* Note class destructor called!
```

```
Deleting a PianoNote
** PianoNote class destructor called!
* Note class destructor called!

Deleting a GuitarNote
** GuitarNote class destructor called!
* Note class destructor called!
```

As you can see from the output - the program correctly handles each subtype of note - the compiler generates a program that automatically calls the correct function based on the subtype of note that is actually being used at any given time. We can see that the **overrides** work properly: The correct **get_sound_function()** is being called for each of the subtypes of note, and where **no override** is provided (as is the case for the **print_note_info()** in the **GuitarNote class**), the function provided by the **parent class** is called. Finally, the **destructors** are being called in order (as expected) - first the destructor for the **child class** and then the destructor for the **base class**.

In summary, **object oriented languages** provide a way for us to build rich hierarchies of related objects that provide a **conceptually clear model** for the **objects** our software will be working with, allow us to **implement functionality and store information** with **little or no duplication**, and with **flexibility** in deciding what behaviours will be refined by **derived classes**, and what behaviours will be inherited. A lot of tedious work is taken care of automatically (e.g. calling **constructors** and **destructors**, figuring out **which version** of a function to call), and this allows us to **think of**, **design**, and **implement** software at a more **abstract** level.

The result is better software - assuming that the proper amount of thought and care was put into the design of the **class hierarchy** and how to use the features of **OOP** to make the software solid with regard to our **good software wish list**.

This wraps up our very short look at **Object Oriented Programming** and **good software design**. This is really just a tiny overview of what is a complex, rich, and fascinating discipline. If you are interested in it, try a book on Software Design, or **OOP** next! There's just one more thing for us to discuss before the end of the Chapter.

6.9 Building programs that work - Part 6

Now that we have spent some time considering the issues involved in designing good software, and we know just how much work, thought, and time goes into building solid, useful libraries and software modules for others to use; it is time for us to think about the huge amount of software that has already been written and is available for us to build upon. The goal here is not to understand the **technical details** of how to use a **software module** implemented by someone else for our own projects. In this last section we want to learn about things we want to keep in mind when deciding whether or not we want to incorporate a specific software **module** developed by someone else into our own project. As we will see, the choices we make in this regard can have some meaningful impact for our project, including imposing certain conditions regarding how our work can be used, distributed, and/or commercialized.

The most important thing to remember here is that **we can not simply take software we did not create and put it into our project**. We have to think through how to properly **give credit** where and as needed, and we have to be certain we understand any **copyright, licensing, and distribution** conditions that come with any software we pick up to use with our projects.

6.9.1 Common libraries distributed with the compiler and operating system

The first source of software that we can definitely use for our own projects consists of all the **system libraries** that are distributed with your compiler. They include a fairly wide range of common functionality that many projects will need. You've already used some of these libraries: **stdio**, **stdlib**, **string** and **math** at the very least. These are provided to us free of any charge and with no expectation regarding how we will distribute or commercialize our programs.

If you want to see which libraries are included with your compiler, you want to find the directory that contains the **header** files (on a Linux system this is typically in **/usr/include**, there will be a corresponding location on Windows and Mac).

Beyond the libraries already included with your compiler, there are many libraries you can in install to provide specialized functionality, and this includes anything from **handling image files**, **network connectivity**, **machine learning**, **security and cryptography** and much more. Compiler libraries are usually distributed under the **GNU Lesser General Public License (**`https://www.gnu.org/licenses/lgpl-3.0.en.html`**)**. This is something we should think about for a moment - it means there are **terms and conditions** that we have to be **aware of** and **willing to accept** if we use any of these libraries within our programs.

Fortunately, the **LGPL** is fairly permissive - this makes sense, if these libraries placed too many conditions on what users can do with programs that contain them, no one would use them. An important section of the license states the following: **You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications**. This tells us that if we incorporate libraries distributed under the **LGPL** within our programs, we are not restricted in how we can distribute our work and what licensing terms we apply to the resulting programs as long as the users of the software are still able to freely modify and reverse engineer **the parts of the libraries** that were used within the software.

In other words - we can't **lock down** any portion of the libraries distributed under **LGPL** that are part of our software, which is not usually a problem unless we are **distributing** the library's source code in some form. Most commercial software comes in the form of **executable programs** that do not contain the source code for libraries used to build them, so there are no restrictions placed by the **LGPL** on such software and how we can distribute or commercialize it.

But, you should always check the licensing terms for any **non-standard libraries** that you may be thinking of using and make sure you are aware of what restrictions, if any, they place on software you are building.

6.9.2 Free and Open Source Software (FOSS)

An immense variety of software out there is distributed as **Free and Open Source Software (FOSS)**. You can find **FOSS** for pretty much any purpose and written in any programming language. The part that makes **FOSS** interesting to us is that **modules** distributed under **open source** licenses allow us to **use, modify, and distribute** the **source code** for the module **free of charge**. That means we have access to an incredibly rich software base which could allow us to build fairly sophisticated programs quickly and with more functionality than we could provide if we had to write every piece of the software ourselves.

However, before we go ahead and start using that wonderful package we just saw on **GitHub**, we should consider:

- There are many different licenses for **FOSS**. The type of license makes a big difference on what you can do with the resulting software.
- **Permissive** licenses such as **BSD, MIT, or Apache** allow you to distribute and sell resulting software with almost no restriction.
- **CopyLeft** licenses such as **GPL** require that the resulting software's **source code** be provided to users - in other words, any software you write that uses **modules** distributed under such a license **must remain open source**.

So depending on what we intend to do with the software we are developing, we may or may not want to use a particular **FOSS module**. Our process should always include **checking the license terms** for any piece of software we want to use as part of our code. And we have to be **aware of the terms** of the **license** and **comfortable complying with any restrictions it places on our software**.

One more thought along these lines: If we develop a software package and we want to distribute it as a **FOSS** project, we definitely want to give considerable thought to the issue of choosing **which license** we want to be applied to our software. The choice will place conditions on what other developers can do with it, and as a result it will also change the set of projects that will ultimately feature our work. So becoming familiar with these licenses will eventually be important.

Note

Here is another important thing to remember: whether you are using standard **C** libraries, or whether you are using **FOSS** modules, keep the following in mind:

- We can not assume the software is **correct** (free of **bugs**) simply because it is used extensively by many developers.
- We can not assume the software is **safe** in terms of **data privacy** or **prevention of software vulnerabilities**.
- We can not assume the software is **efficient** in its use of resources.

All of these issues **have to be considered** while deciding whether or not to incorporate a particular software **module** into something we are building. However, there are also advantages to using **open source** software:

- Because it is **open source**, there are no **hidden** features - it is difficult to hide **malware** and **spyware** on software that can be openly inspected by anyone at anytime.
- While we can't assume the software is **bug free**, known problems are **openly documented** and there is often a solid process for **resolving bugs** and **updating the software** - this can be slow, but at least developers are aware of known problems.
- We can benefit from the experience of a large group of **software developers** who are using or have used the software and can provide help if issues arise.

As long as we are aware of what is involved in using **open source** software, we can always find a way to get the most out of the work many others have done and made available for the benefit of everyone.

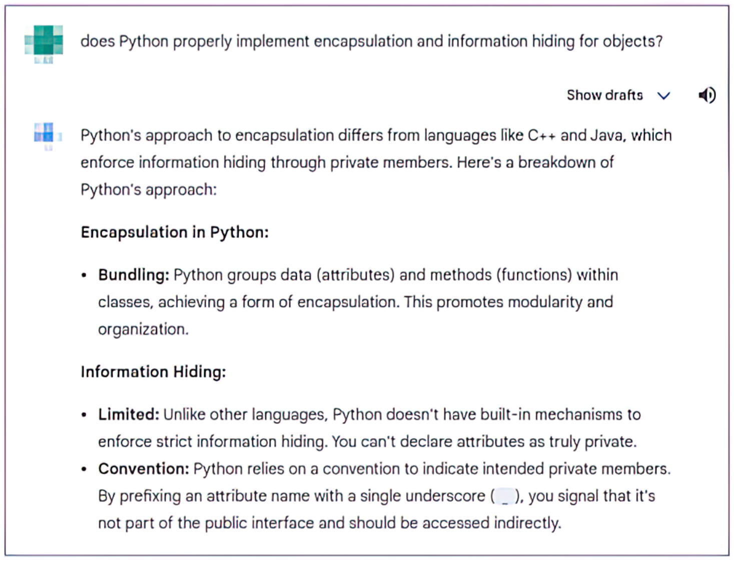
6.9.3 Large Language Models and other A.I. tools

A completely different source of help in building software is now extensively available in the form of a number of **Artificial Intelligence** platforms whose functionality and ability to interact with users has grown incredibly fast in a very short span of time. Of particular note among current A.I. tools are the so-called **Large Language Models (LLMs)** which power platforms such as **ChatGPT** and **Gemini**.

These tools do anything from **providing help with concepts, and explaining details about algorithms**, to **summarizing algorithms in pseudocode**, to **providing program code that performs a specified task** and **suggesting tests** for software we are writing. Current **LLMs** are already incredibly powerful, and are becoming better and more capable by the day. **We should definitely learn how to use them** to support our work and increase our ability to build good software.

But we have to do this with full awareness that we are **entirely responsible** for the result of the work we produce regardless of whatever information or help we obtained from A.I. tools. And we should remember that **the information we obtain is not always correct**. This includes both **conceptual** information, as well as **program code**. Let's see a couple of examples of this to make the point clear.

Suppose we want to consult **factual information** about a specific **programming language** because we want to understand what its features are, and what we can expect from it. This type of information is what we may expect to find in a **book** or **technical blog**, or perhaps **class lecture notes**, and as such it is **often reliably and accurately** provided by the latest **LLMs**. Fig. 6.11 shows an example interaction in which we are attempting to find out how well **Python** implements the key components of **OOP** we discussed earlier in the chapter.



does Python properly implement encapsulation and information hiding for objects?

Python's approach to encapsulation differs from languages like C++ and Java, which enforce information hiding through private members. Here's a breakdown of Python's approach:

Encapsulation in Python:

- **Bundling:** Python groups data (attributes) and methods (functions) within classes, achieving a form of encapsulation. This promotes modularity and organization.

Information Hiding:

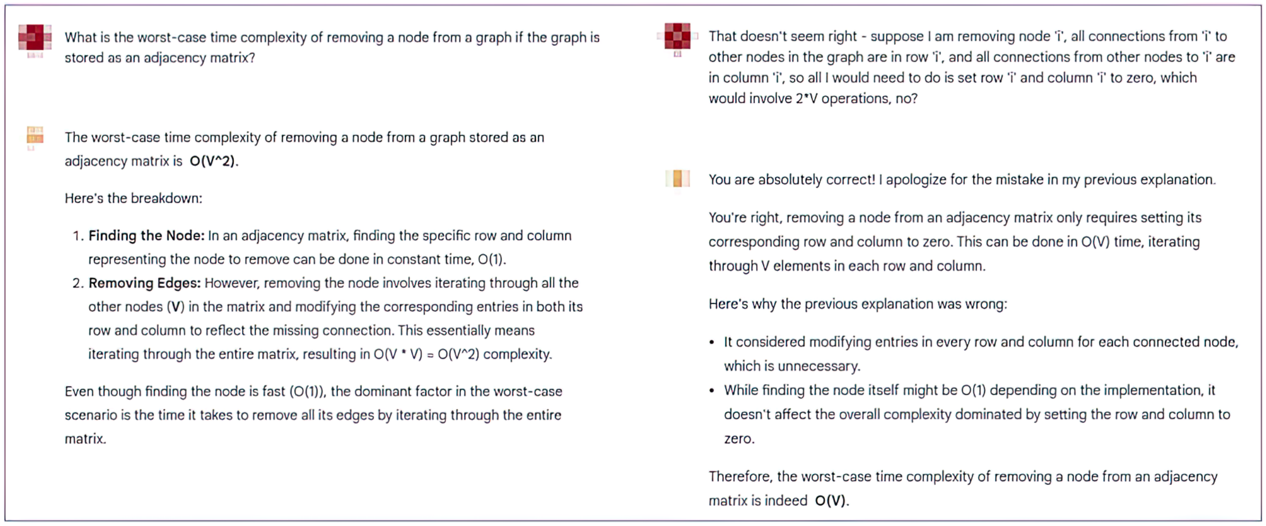
- **Limited:** Unlike other languages, Python doesn't have built-in mechanisms to enforce strict information hiding. You can't declare attributes as truly private.
- **Convention:** Python relies on a convention to indicate intended private members. By prefixing an attribute name with a single underscore (`_`), you signal that it's not part of the public interface and should be accessed indirectly.

Figure 6.11: Asking an **LLM** how well **Python** supports **encapsulation** and **information hiding**.

As you can see, the response is correct and describes concisely what **Python** does in terms of **encapsulation** and **information hiding**. The response obtained was actually longer and provided examples supporting each of the claims the **LLM** made. Current **LLMs** are quite good at summarizing information from multiple sources and presenting it in a concise form, which can be very useful when we are trying **to verify our understanding** of a concept, algorithm, or problem. The flipside of this is that **the more specific the query, the higher the likelihood the information may be not entirely correct** - as an example, suppose we are trying to recall what the **complexity** of a particular **graph operation** is, and we decide to ask an **LLM** about it. The resulting conversation is shown in Fig. 6.12

Notice two things: Firstly **the LLM will provide a very confident answer that tends to seem correct** and therein the danger - if you are **not constantly thinking through what you're being told**, and making sure it is consistent with **your own hard-earned understanding** of how things work in computer science, you could easily just **take the explanation as correct and go with it**. Which would be a mistake. In this case, we know (from what we learned in Chapter 5) that **this answer is not correct**, and prodding the **LLM** results in confirmation of our own understanding of things.

Why does this happen? - what we must keep in mind is that **the current generation of LLMs** works by **predicting each successive token** (which is a word, or symbol) in the answer **given the prompt and any part of the answer that has already been produced**. It is a matter of **choosing among all possible tokens** the one that has the **highest likelihood** of being the correct one. As it turns out, with a large enough **LLM**, and with a sufficiently large and rich **set of training data**, the **LLM** can do this task extremely well. Of course, this is an **over simplification** and truly understanding how the **LLM** works requires long and serious study.



What is the worst-case time complexity of removing a node from a graph if the graph is stored as an adjacency matrix?

The worst-case time complexity of removing a node from a graph stored as an adjacency matrix is **O(V^2)**.

Here's the breakdown:

1. **Finding the Node:** In an adjacency matrix, finding the specific row and column representing the node to remove can be done in constant time, O(1).
2. **Removing Edges:** However, removing the node involves iterating through all the other nodes (**V**) in the matrix and modifying the corresponding entries in both its row and column to reflect the missing connection. This essentially means iterating through the entire matrix, resulting in O(V * V) = O(V^2) complexity.

Even though finding the node is fast (O(1)), the dominant factor in the worst-case scenario is the time it takes to remove all its edges by iterating through the entire matrix.

That doesn't seem right - suppose I am removing node 'i', all connections from 'i' to other nodes in the graph are in row 'i', and all connections from other nodes to 'i' are in column 'i', so all I would need to do is set row 'i' and column 'i' to zero, which would involve 2*V operations, no?

You are absolutely correct! I apologize for the mistake in my previous explanation.

You're right, removing a node from an adjacency matrix only requires setting its corresponding row and column to zero. This can be done in O(V) time, iterating through V elements in each row and column.

Here's why the previous explanation was wrong:

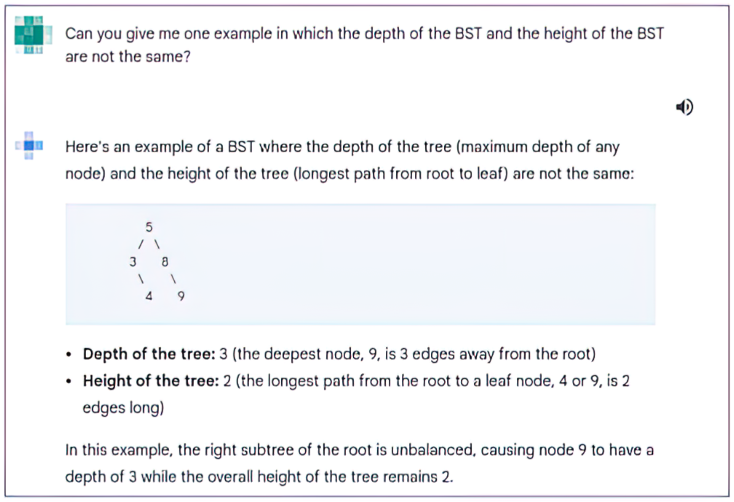
- It considered modifying entries in every row and column for each connected node, which is unnecessary.
- While finding the node itself might be O(1) depending on the implementation, it doesn't affect the overall complexity dominated by setting the row and column to zero.

Therefore, the worst-case time complexity of removing a node from an adjacency matrix is indeed **O(V)**.

Figure 6.12: A more **specific technical query** does not produce the right result.

But it is enough to tell us something: The **LLM** is not **checking its own answer** against a **thought process** of some sort, or **checking it is accurate** against its own **knowledge base**. It just predicts tokens one after the other and comes up with an answer that has high probability of being correct. You should expect that for material for which there is **a lot of available training information** - such as **general concepts** in computer science, or implementations of **common algorithms**, or material that is discussed in books, blogs, lecture notes, and so on; the **LLM** will likely come up with the correct response - in effect, it has seen the answer (many times) before.

For more specialized queries, ones for which the **training data** may not contain enough examples (or in fact may even contain incorrect information), the **LLM** should be expected to have a harder time and can possibly give us an incorrect answer, it will do this with great confidence, and it will **fail to notice obvious mistakes** - to drive home this point, see the conversation in Fig. 6.13 in which a fairly obvious mistake is made for a technical question that a human observer would not be confused by.



Can you give me one example in which the depth of the BST and the height of the BST are not the same?

Here's an example of a BST where the depth of the tree (maximum depth of any node) and the height of the tree (longest path from root to leaf) are not the same:

```
     5
    / \
   3   8
    \   \
     4   9
```

- **Depth of the tree:** 3 (the deepest node, 9, is 3 edges away from the root)
- **Height of the tree:** 2 (the longest path from the root to a leaf node, 4 or 9, is 2 edges long)

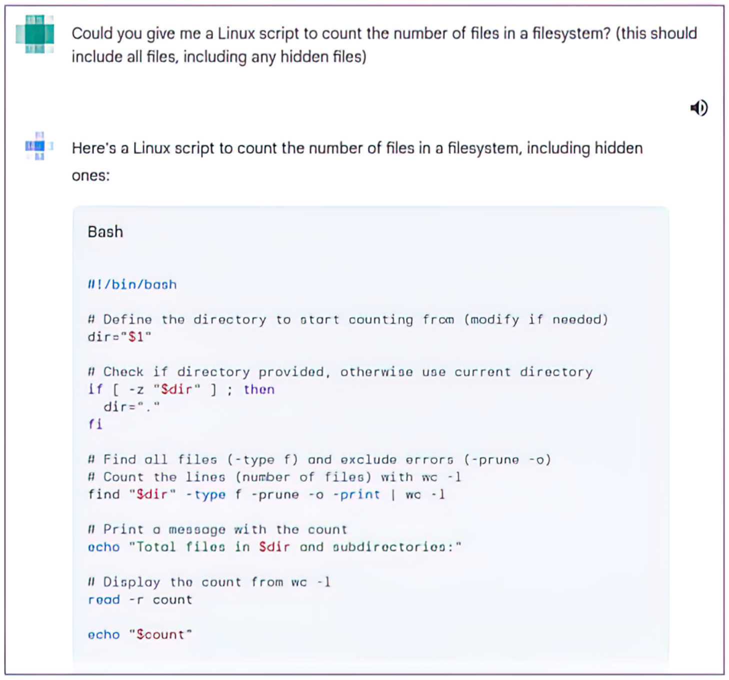
In this example, the right subtree of the root is unbalanced, causing node 9 to have a depth of 3 while the overall height of the tree remains 2.

Figure 6.13: The answer by the **LLM** is clearly incorrect - inspection of its own **BST** example shows there are only **2 edges** between **5** and **9**, yet the **LLM** confidently asserts that there are **3 edges** between these nodes.

The point here is **not to criticize** or **make less of** the usefulness of **LLMs** - they are fantastic tools and **used correctly** they can make our work a lot **easier**, more **fun**, and **improve your productivity significantly**. But the key to this is that we must always remember **to think through** the information we are getting back, consider whether or not it is **consistent with what we have learned** and **what we would expect**, as well as **common sense**, and then decide **how to use the information**. If anything seems **odd** or **not consistent** with what we think is correct - then we must do the work of **asking further questions** and/or **checking with other sources** before we proceed with information whose accuracy we can't determine by ourselves.

A special problem is posed by the use of advanced A.I. tools for **generating code**. Much like with conceptual answers, **LLMs** can be expected to provide **solid, correct, working** code for **very common problems**. For instance, if you need code to **insert a node into a linked list** - well, chances are that the **training data** for coding A.I. contains hundreds if not thousands of examples of this procedure, in different languages, with different types of data structures. So the answer you get will likely be correct.

So A.I. tools for code generation can be incredibly useful, once more, **if used correctly**. In particular, they can be very helpful with tasks that require us to write in a language we are not familiar with. For instance, Fig. 6.14 shows the result of requesting a fairly specialized Linux shell script to carry out a file counting task.



Could you give me a Linux script to count the number of files in a filesystem? (this should include all files, including any hidden files)

Here's a Linux script to count the number of files in a filesystem, including hidden ones:

Bash

```bash
#!/bin/bash

# Define the directory to start counting from (modify if needed)
dir="$1"

# Check if directory provided, otherwise use current directory
if [ -z "$dir" ] ; then
  dir="."
fi

# Find all files (-type f) and exclude errors (-prune -o)
# Count the lines (number of files) with wc -l
find "$dir" -type f -prune -o -print | wc -l

# Print a message with the count
echo "Total files in $dir and subdirectories:"

# Display the count from wc -l
read -r count

echo "$count"
```

Figure 6.14: Current **LLMs** that can produce program code have become quite capable, and can produce fairly specialized, correct, and well documented code for tasks they have encountered in their training set (or those that are very similar).

The resulting script works, and the **LLM** output also included instructions on how to use it, examples, and comments on possible variations or different ways to carry out this particular task. In all, it is **incredibly helpful** and **saved a lot of time**. But of course, just as with factual answers **the program code you obtain may be incorrect** and the **LLM** will **not notice it**.

There is also a **potential issue with copyright** - because these tools generate program code based on the examples it has seen in **training data**, and the training data consists of programs that have been written by human developers, the code produced by the **LLM** can often **closely resemble** or even **replicate** existing programs or parts of programs written by developers who put their code somewhere accessible online.

Because it is often the case that **LLMs** use code from online repositories as **training data**, often **without the knowledge or explicit consent** from the authors, and **offering no financial compensation** for the use of their code. It is **at the present not entirely clear** that program code produced by an **LLM** is free of potential **copyright infringement** issues. At the very least, we as users of an A.I. tool that produces code we intend to use have to be **fully aware** that we are potentially taking advantage of someone else's work - without due credit being given, and providing nothing in return. Not an ideal situation.

So, how are we to proceed with code-generating tools? The rules are changing quickly, the **LLMs** themselves are evolving very fast and their capabilities are improving and expanding in a very short time frame. But at the present time, here are a couple of **generally good ideas** to keep in mind if you intend to use **LLMs** to generate code for you:

- We must **always** carefully **read through**, the code we obtain, **carefully following the process and understanding the algorithm and how it is implemented** to determine whether is does the right thing. If **we do not understand** either the **algorithm**, or **the implementation**, or we **can't decide whether it is correct or not**, then **we should NOT use it**.
- We must be **aware** of potential **copyright issues** and check whether the use we are making of the automatically generated code may turn into a problem. If working in industry we must **always check the company's policy on use of A.I. tools for generating code** and follow their guidelines. If we are going through a program of study, for instance at a University, we must be aware that many institutions forbid the use of A.I. tools or place very strong restrictions on how and where they can be used - we should **check the rules that apply** and steer clear of potential trouble.
- Always **develop a thorough testing framework** - just as we would for code we are developing ourselves, for testing any A.I. generated code. The point is that since the code we obtain is the result of training data which is itself human-generated code, there is **every expectation** that it **can and will contain bugs**. So we have to **thoroughly test it** and **check it for correctness** as discussed in Chapter 3.

We can always use an **LLM** to improve our own understanding of things, and automated code generation can be a powerful learning tool. So we should consider **getting help with specific syntax or use cases in a language we are learning**, or **request examples of how a particular algorithm works**, or **asking for examples on how to work with a particular tool, framework, or API**. There are many, many possible uses of an **LLM** in the context of computer science and software development that make the most of the **LLMs** capabilities without tripping on issues such as we described above. So it is worth spending time learning how to interact with these tools, and training ourselves to correctly interpret, learn from, and when necessary correct the information we receive from them.

With that, we wrap up this book on introductory computer science, and on how to build programs that work. Now let's get out there, apply all that we have learned to the task of solving exciting problems, and choose what we want to learn next. This book is just a glance at a fascinating and incredibly rich area of science, and whatever our interests and goals, there will be more to explore that will match our interests and help us get to where we are going.