

# 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 **if** 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 data** 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.

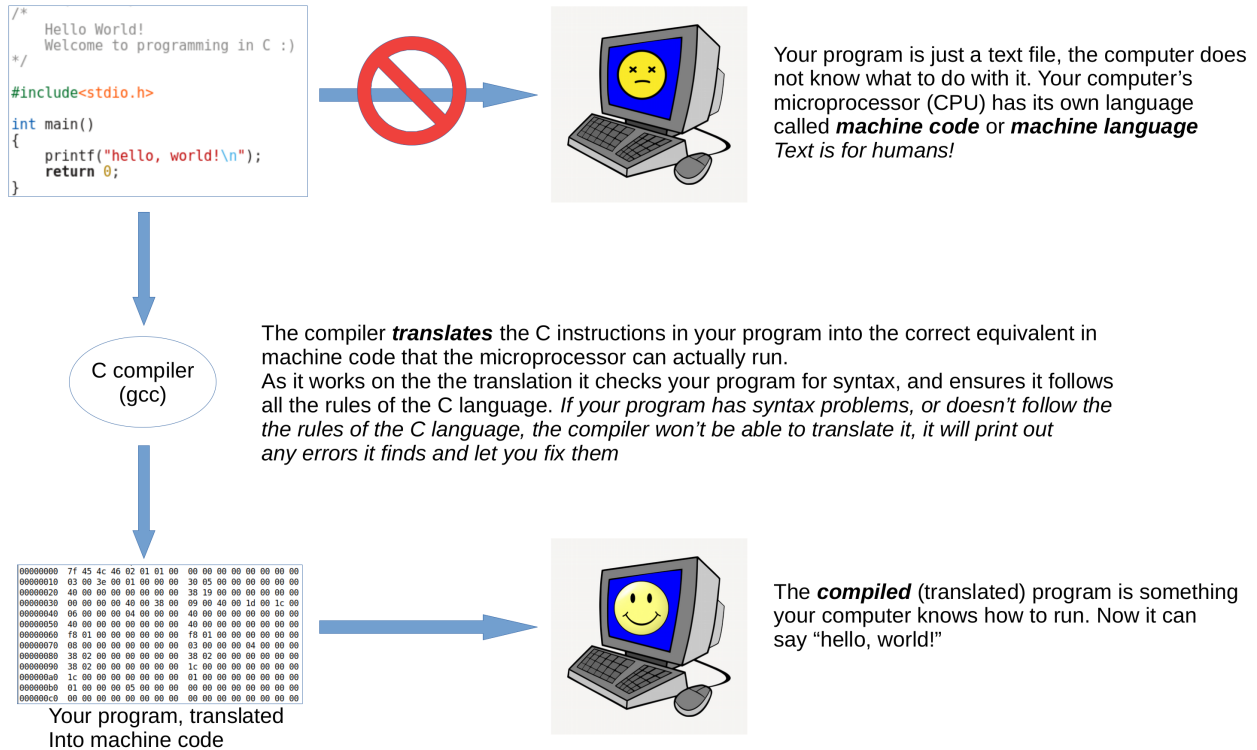
To compile your code:

- 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.





**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 formatted 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:

<code>a==b</code>	True <b>if</b> a equals b
<code>a&gt;b</code>	True <b>if</b> a is strictly greater than b
<code>a&lt;b</code>	True <b>if</b> a is strictly lesser than b
<code>a&gt;=b</code>	True <b>if</b> a is greater than or equal to b
<code>a&lt;=b</code>	True <b>if</b> a is lesser than or equal to b
<code>a!=b</code>	True <b>if</b> 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:

<code>  </code>	Logical or (this is a <b>double</b> vertical bar)
<code>&amp;&amp;</code>	Logical and (this is a <b>double</b> ampersand)
<code>!</code>	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.