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**:

```
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

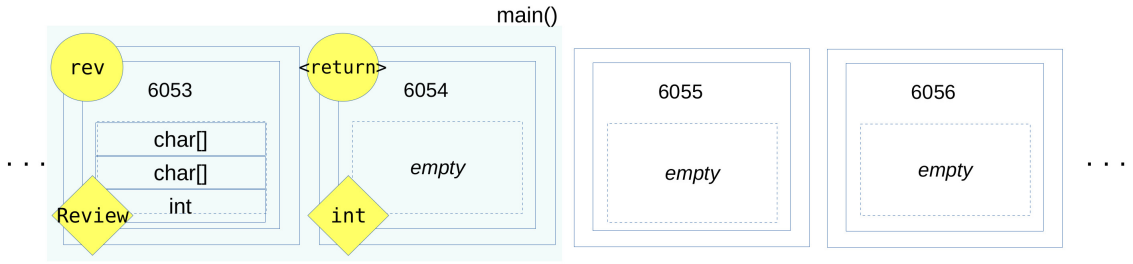

Figure 3.3: Memory model for the program in Exercise 3.3. Note that there is only **2 boxes**, one for the **Review** variable called **'rev'** which is of compound type, and one for **main()'s return value**

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.

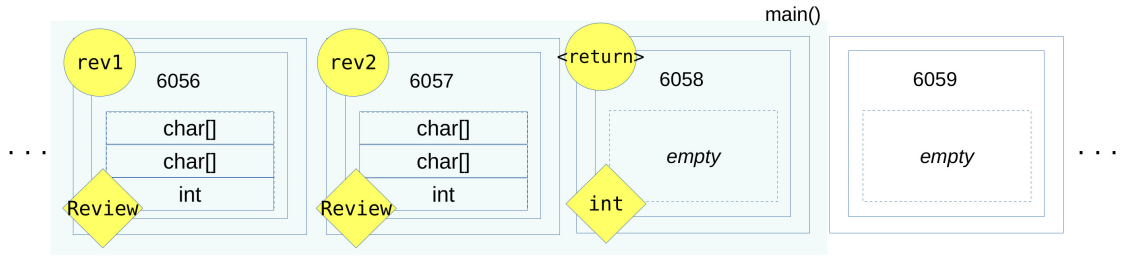

Figure 3.4: Memory model for the program in Example 3.4 just after memory has been reserved for **main()**.

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

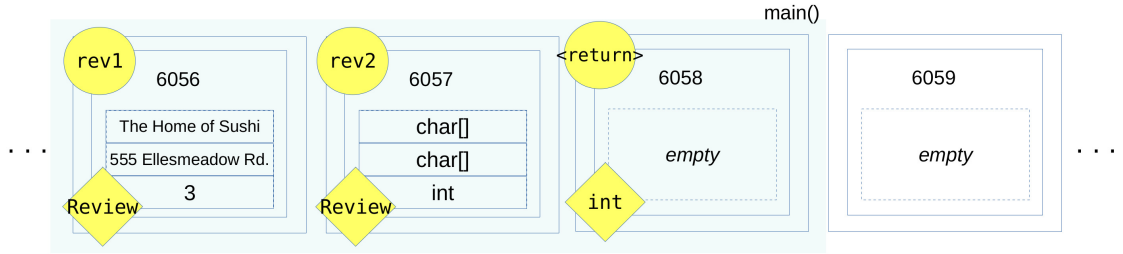

Figure 3.5: Memory model for the program in Example 3.4 after filling-in information for **rev1**.

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.

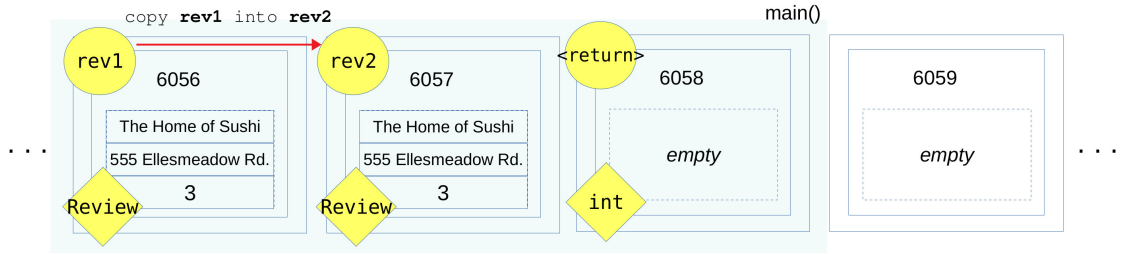

Figure 3.6: Memory model for the program in Example 3.4 after the instruction **'rev2=rev1;'**.

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

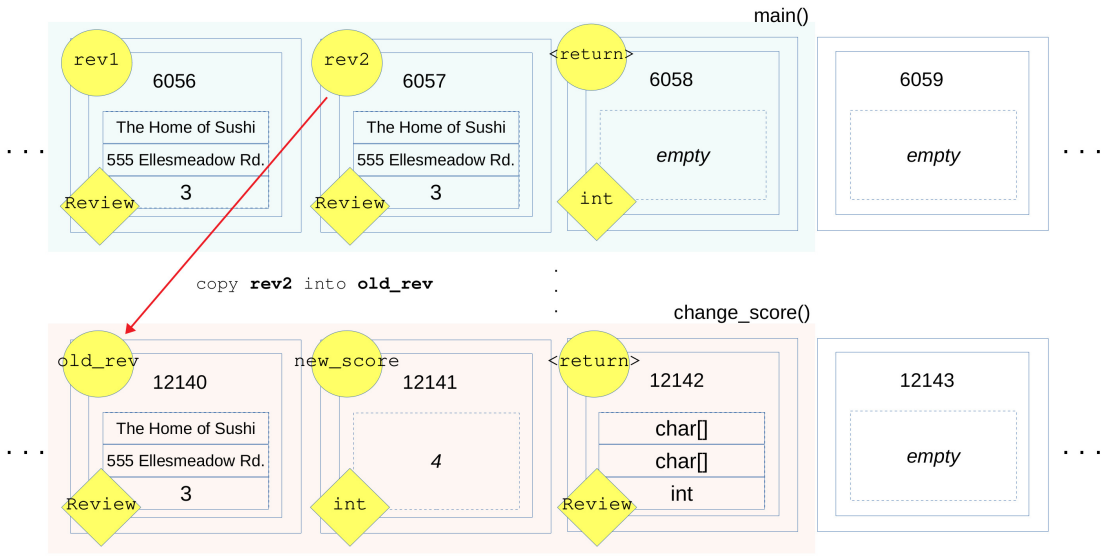

Figure 3.7: Memory model for the program in Example 3.4 after the instruction **'rev2=new_score(rev2,4);'** has reserved space for the function's input arguments and return value, and copied the input arguments into their corresponding boxes.

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

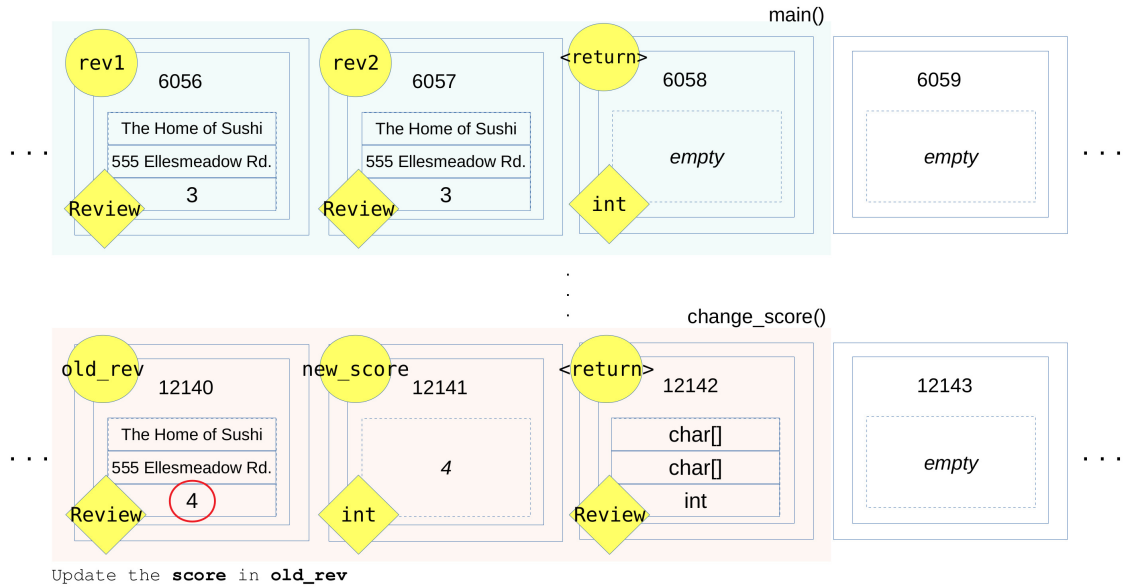

Figure 3.8: Memory model for the program in Example 3.4 after updating the score for the **old_rev** CDT.

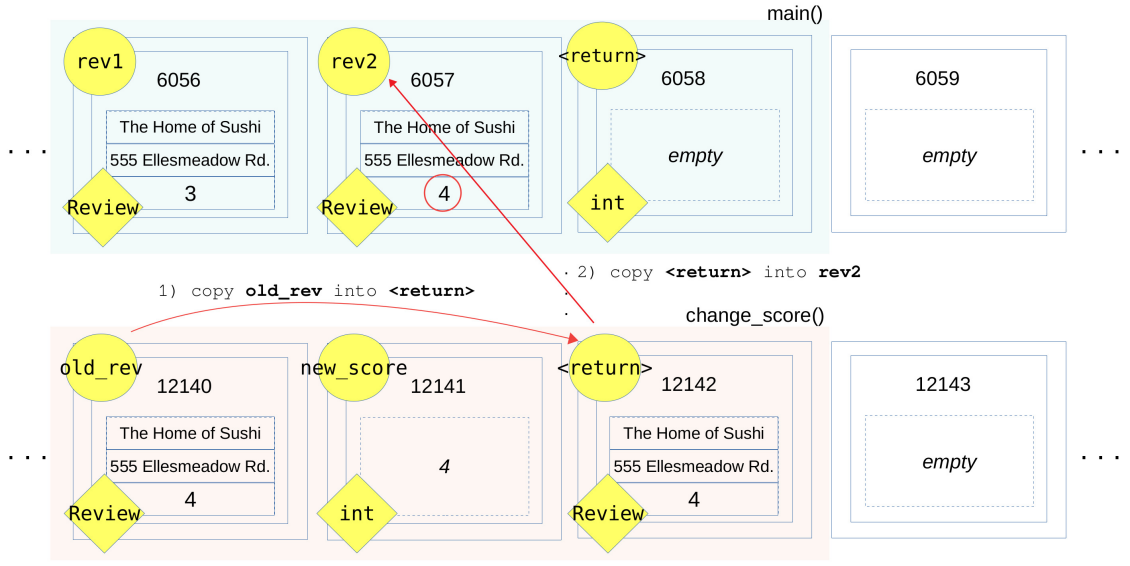

Figure 3.9: Memory model for the program in Example 3.4 showing the effect of the function's **return** statement.

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.

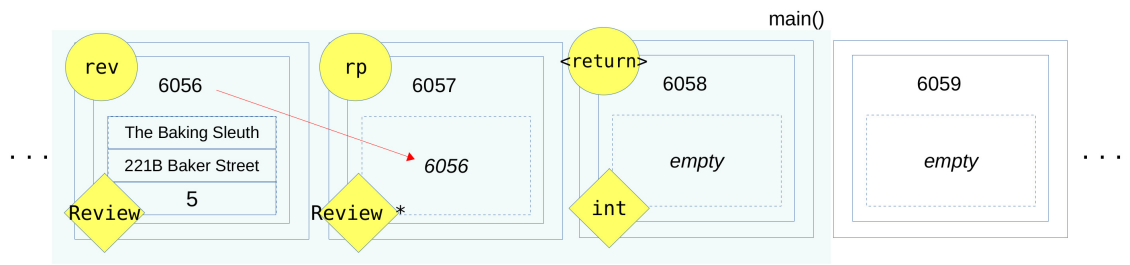

Figure 3.10: Memory model for the program in Example 3.5 after the line **'rp=&rev;'**.

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.

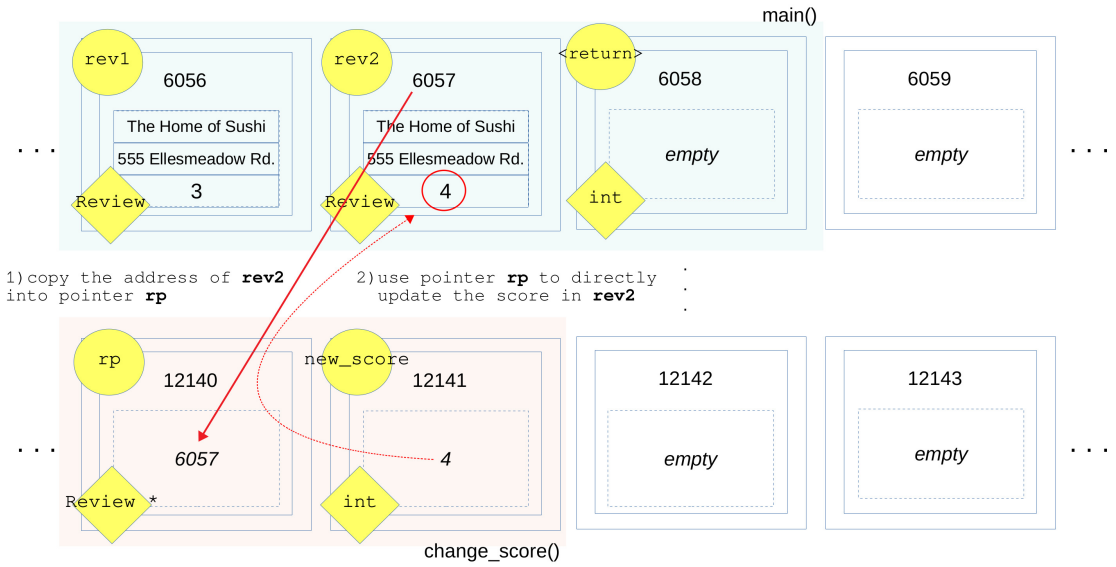

Figure 3.11: Memory model for the program in Example 3.6 after the call to **change_score()** has been set up.

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.

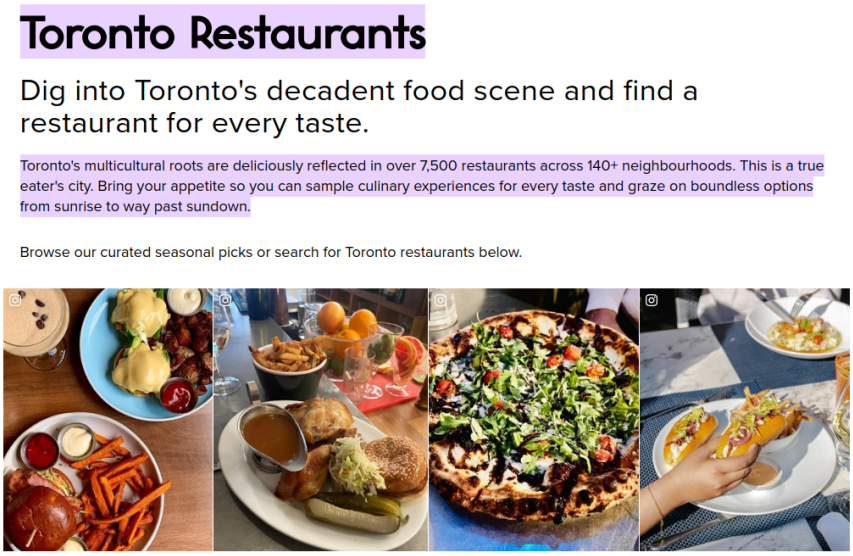

Figure 3.12: Toronto is a good place to be, if you like food!. *Image from: https://www.destinationtoronto.com/restaurants/*

- 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

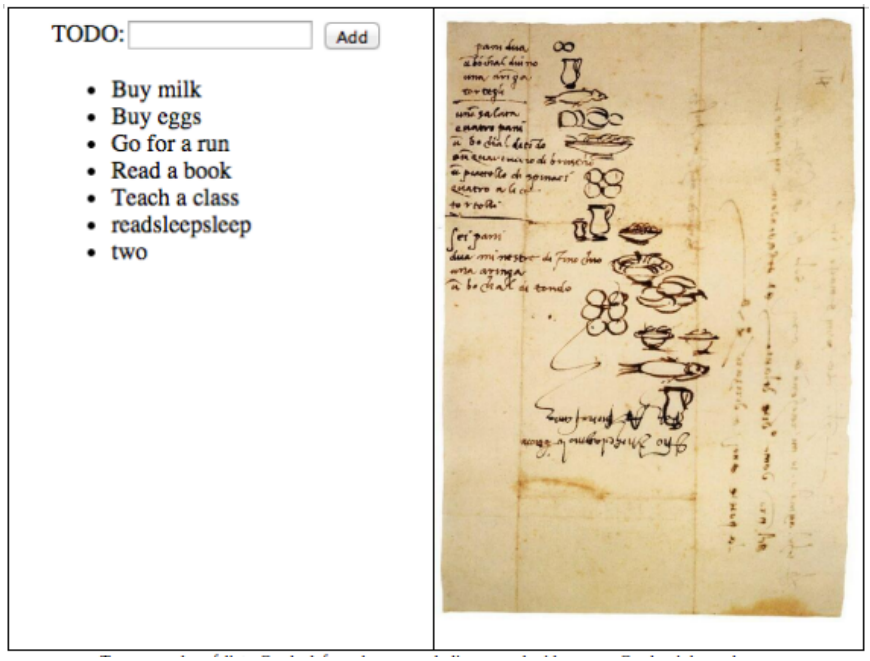

Figure 3.13: Two examples of lists. On the left we have a to-do list created with an app. On the right we have Michelangelo's shopping list from the 16th century. *Images: (left) Lubaouchan, Wikimedia Commons, CC-SA 4.0; (right) Michelangelo, Wikimedia Commons, Public Domain.*

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.

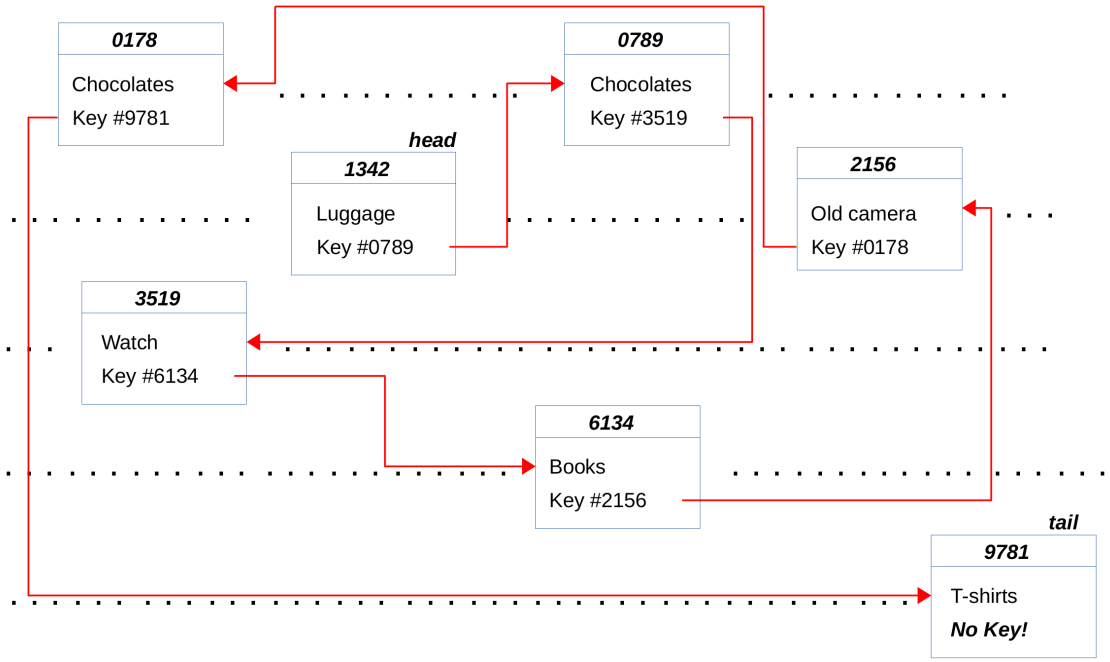

Figure 3.15: Map of the locker room at the train station after organizing all the items stored there into a **linked list**.

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.

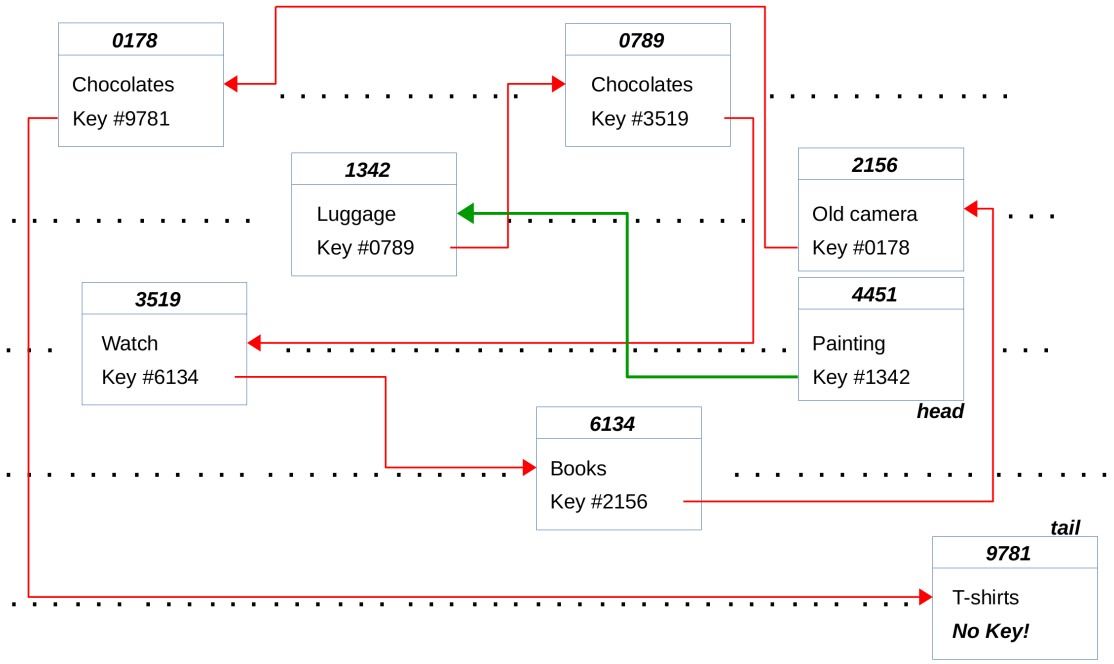

Figure 3.16: Map of the locker room after we add an item (a painting) to our collection **at the head** of the list.

.

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.

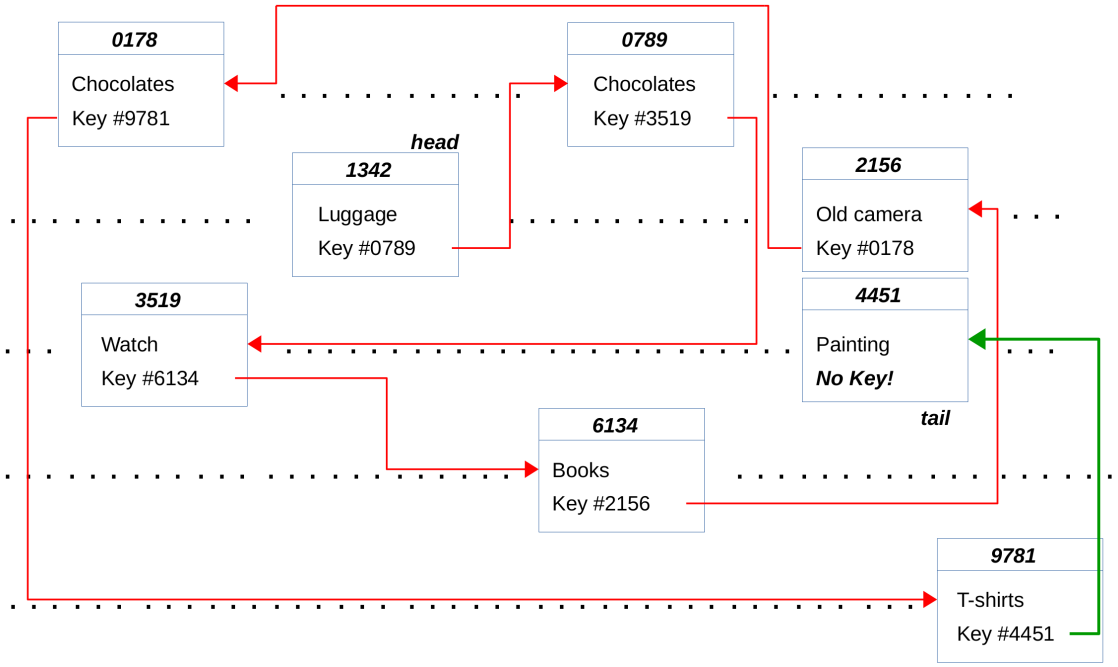

Figure 3.17: Map of the locker room after we add an item (a painting) to our collection **at the tail** of the list.

.

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) → **#4451** (painting) → **#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

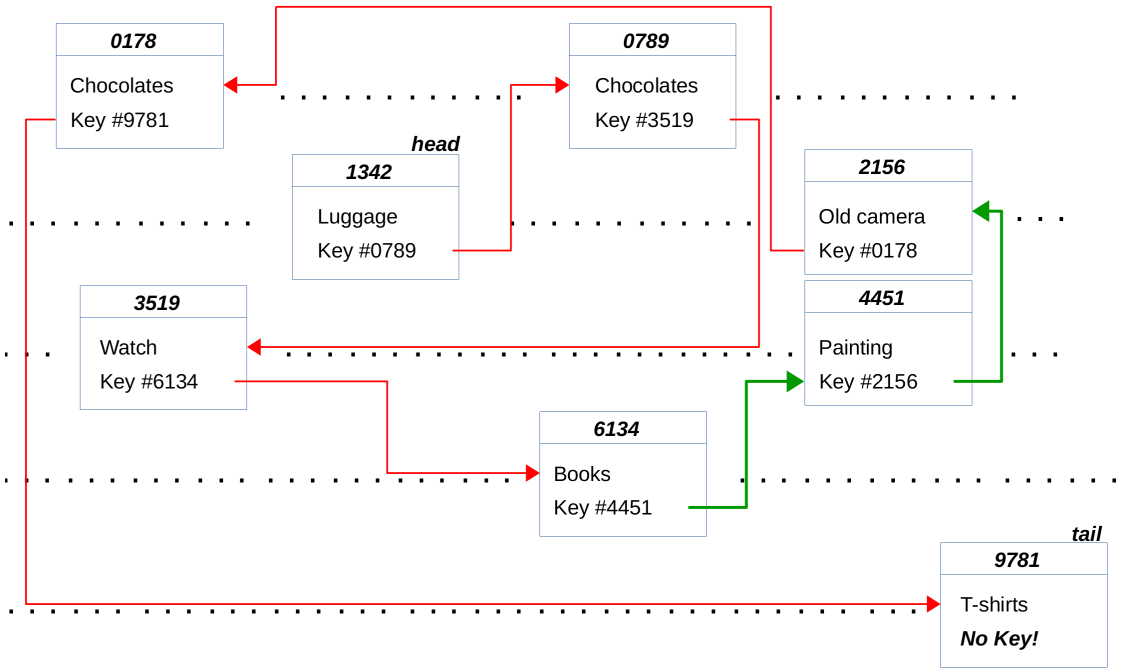

Figure 3.18: Map of the locker room after we add an item (a painting) to our collection **in between two other items** of the list.

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**) → **#0789** (no items, key for **#3519**) → **#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**) → **#3519** (watch) ... (rest of the list). This is shown in Fig. 3.19.

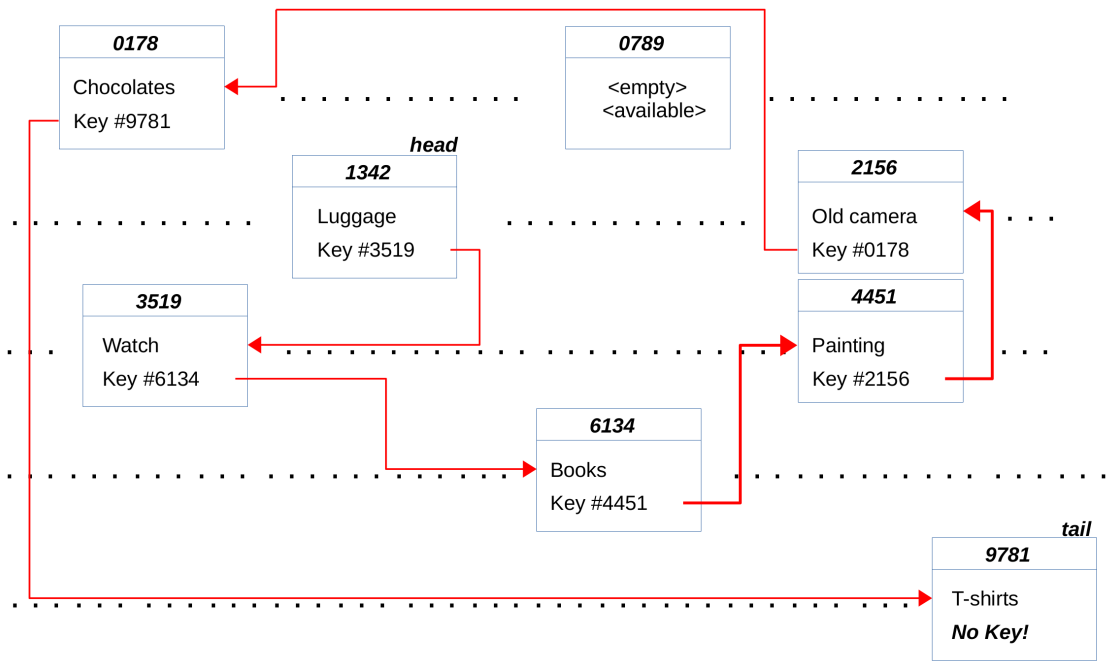

Figure 3.19: Map of the locker room after we remove the chocolates from our collection.

.

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

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.

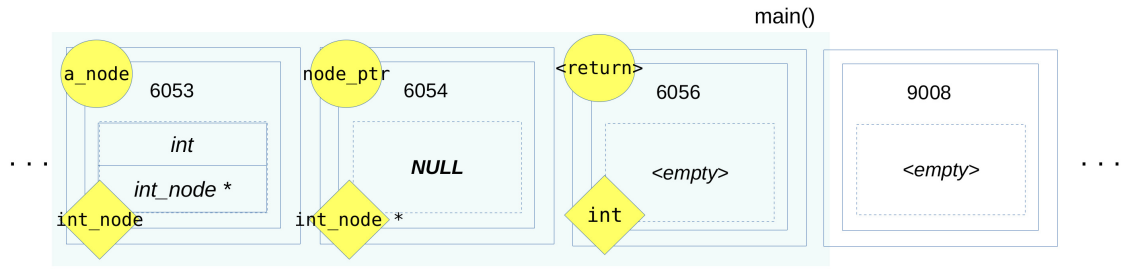

Figure 3.20: Memory model for the code in Example 3.13 just after space has been reserved for the program.

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);
```

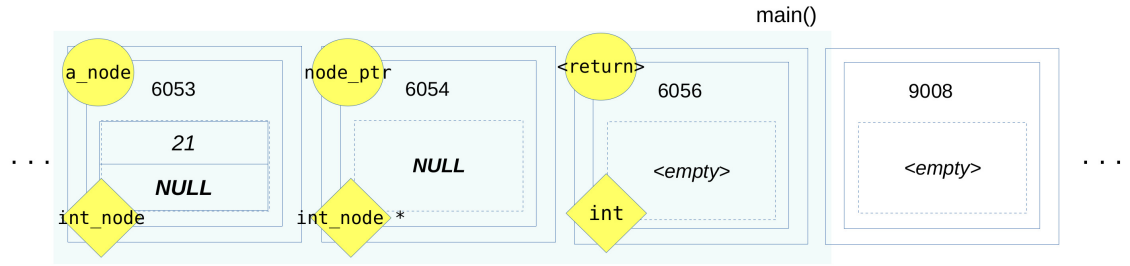

Figure 3.21: Memory model for the code in Example 3.13 after **a_node** has been filled with information.

.

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.

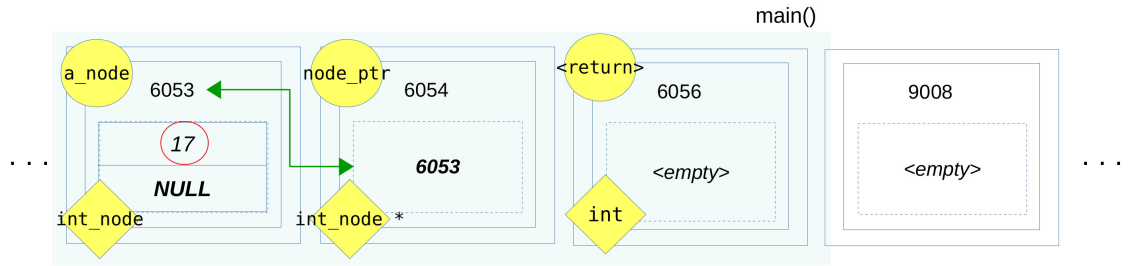

Figure 3.22: Memory model for the code in Example 3.13 just before the program ends.

.

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
```

```
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.

```
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:

```
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.

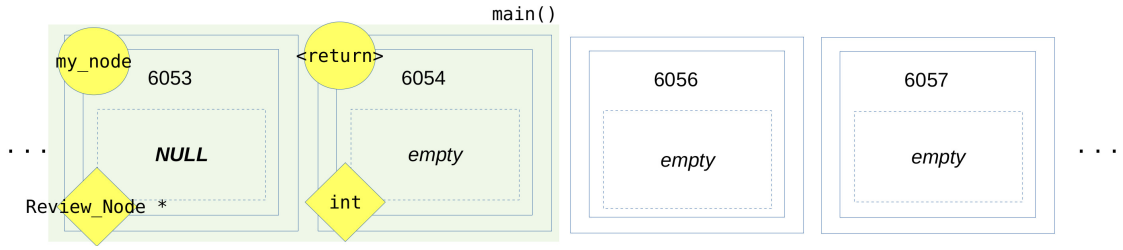

Figure 3.23: Memory model for the code in Example 3.14 just after space is reserved for **main()'s** variables.

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.

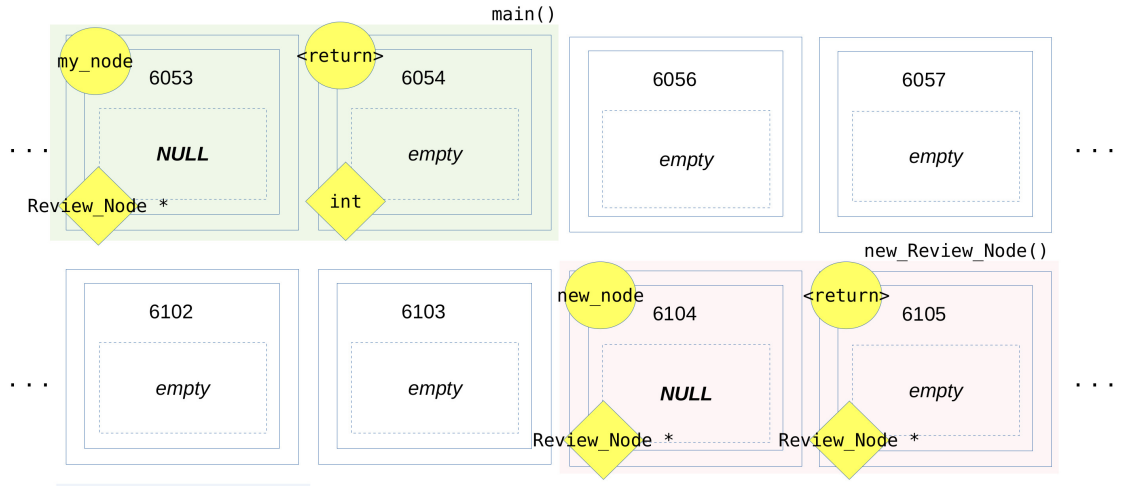

Figure 3.24: Memory model for the code in Example 3.14 just after the call to **new_Review_Node()** has reserved space for the function's variables.

.

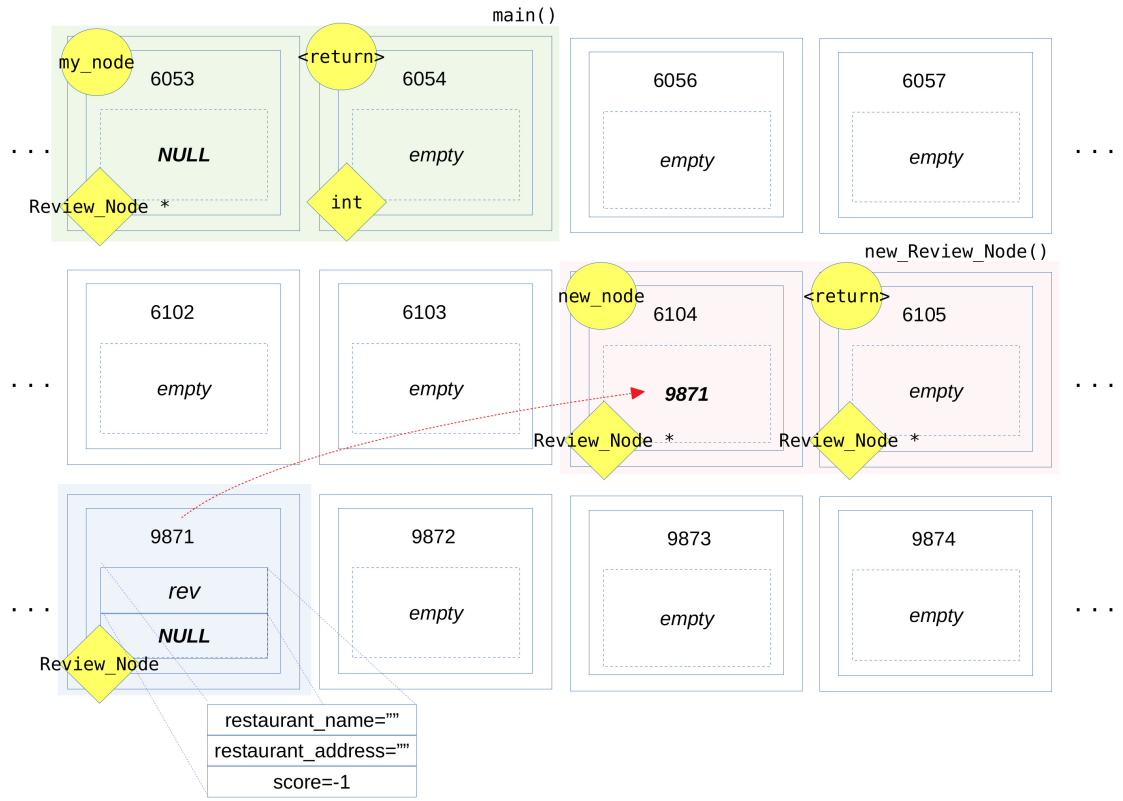

Figure 3.25: Memory model for the code in Example 3.14 after the function **new_Review_Node()** has reserved memory for a new **Review_Node** and filled it with **default** values.

.

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.

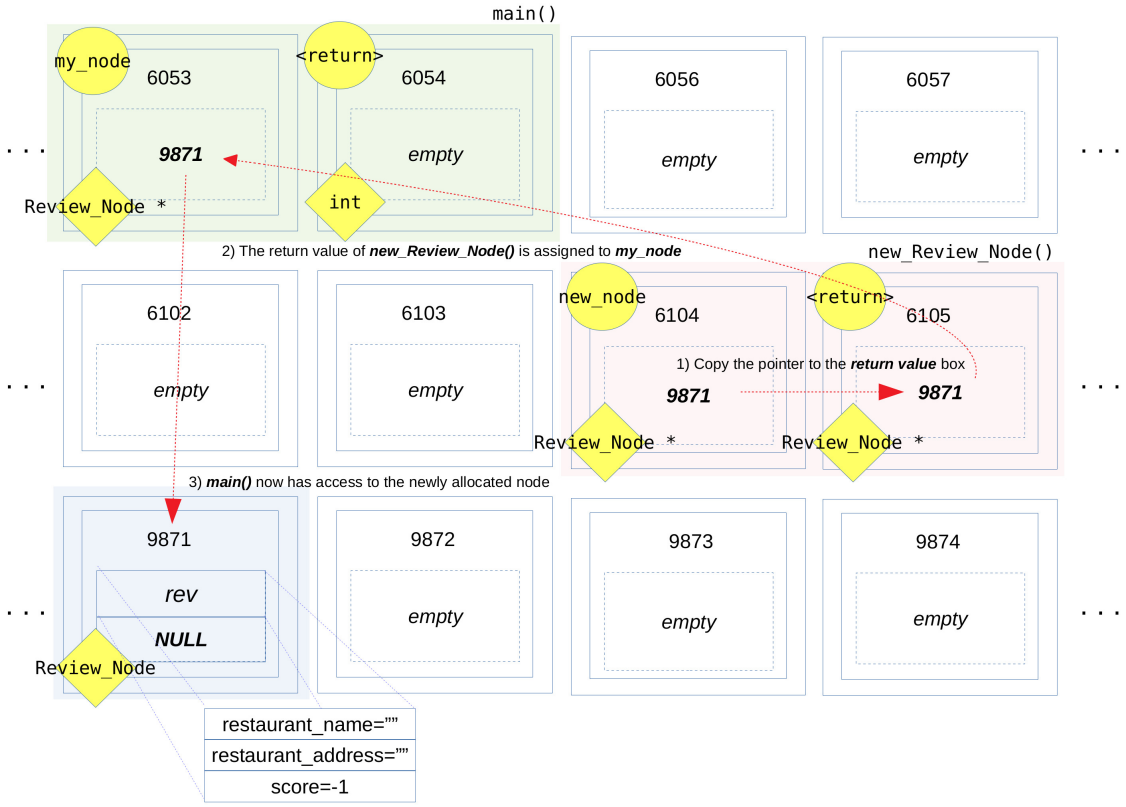

Figure 3.26: Memory model for the code in Example 3.14 once the **return statement** copies the pointer to the newly allocated node back to **main()'s** pointer variable **'my_node'**.

.

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

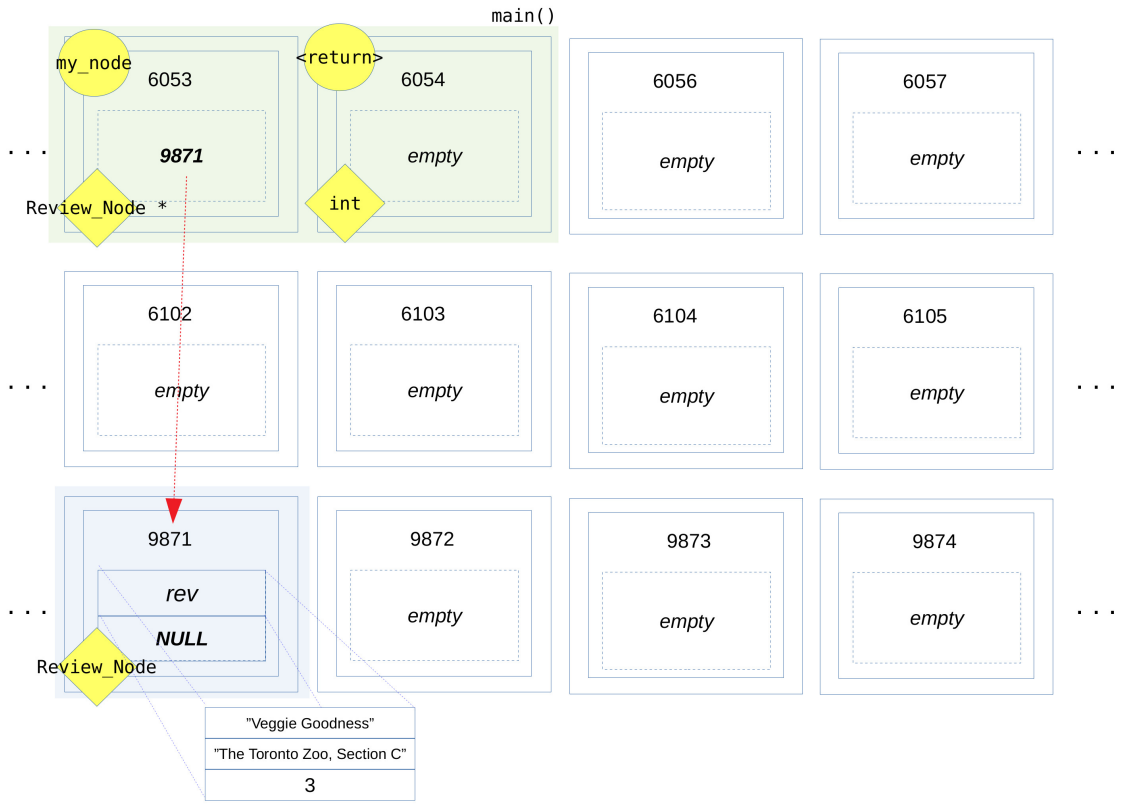

Figure 3.27: Memory model for the code in Example 3.14 after **main()** has filled-in the new node with information.

.

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.

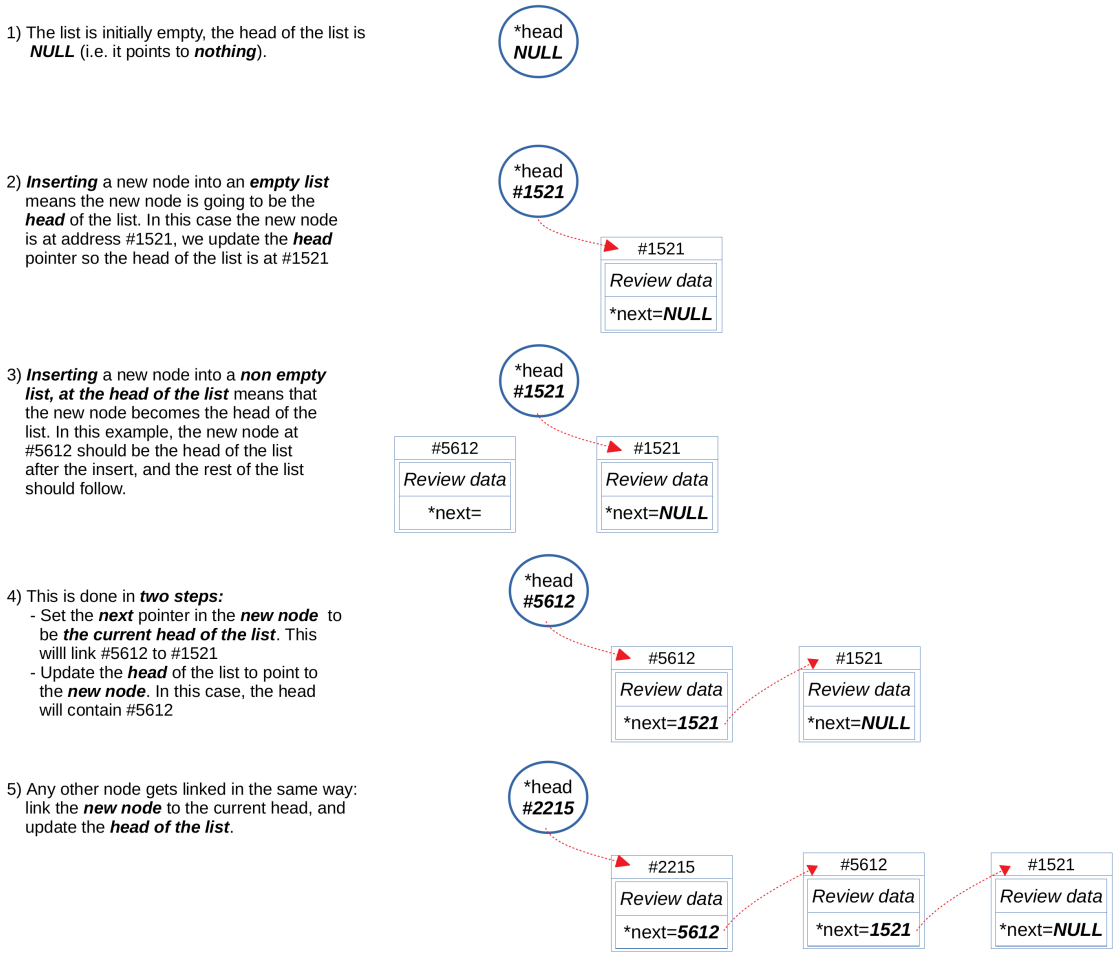

Figure 3.28: Sequence showing how the linked list grows as nodes are added **at the head** of the list.

.

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.

```
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?

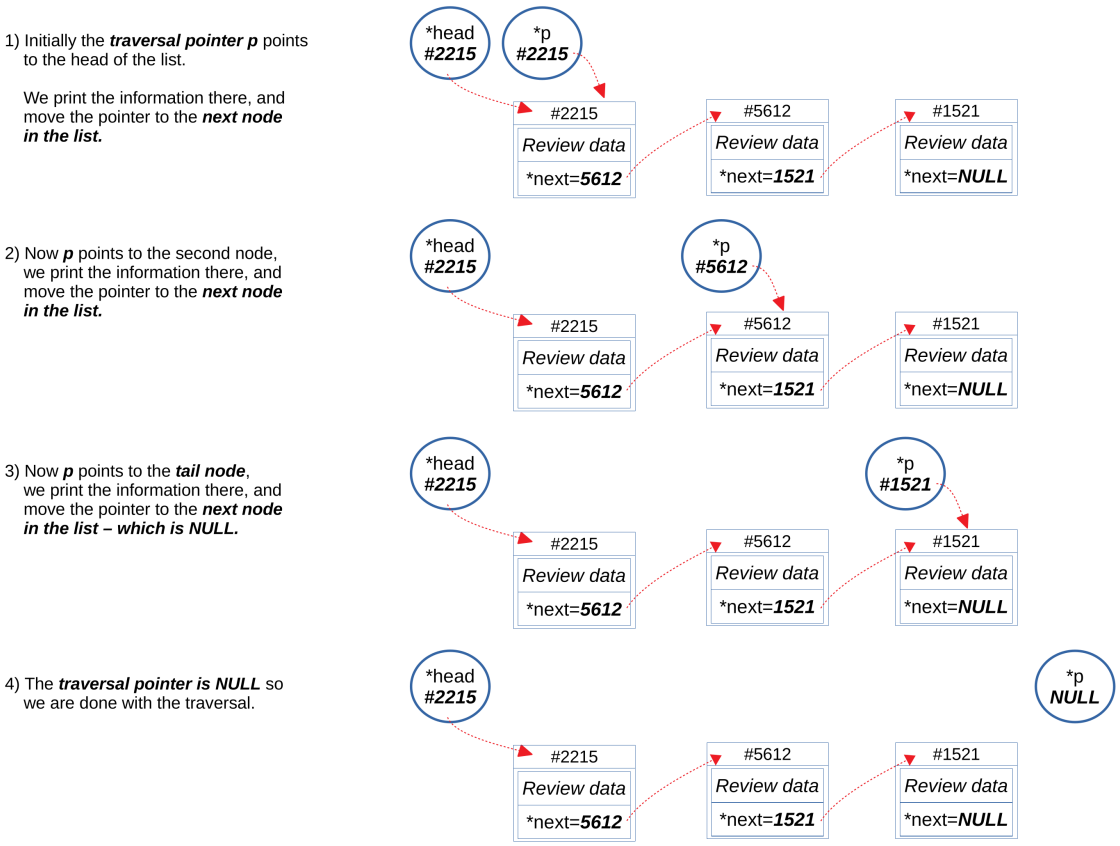

Figure 3.29: Sequence showing how the **traversal pointer p** moves down the linked list, visiting each node in sequence until reaching the end of the list (indicated by a **NULL** pointer).

.

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:

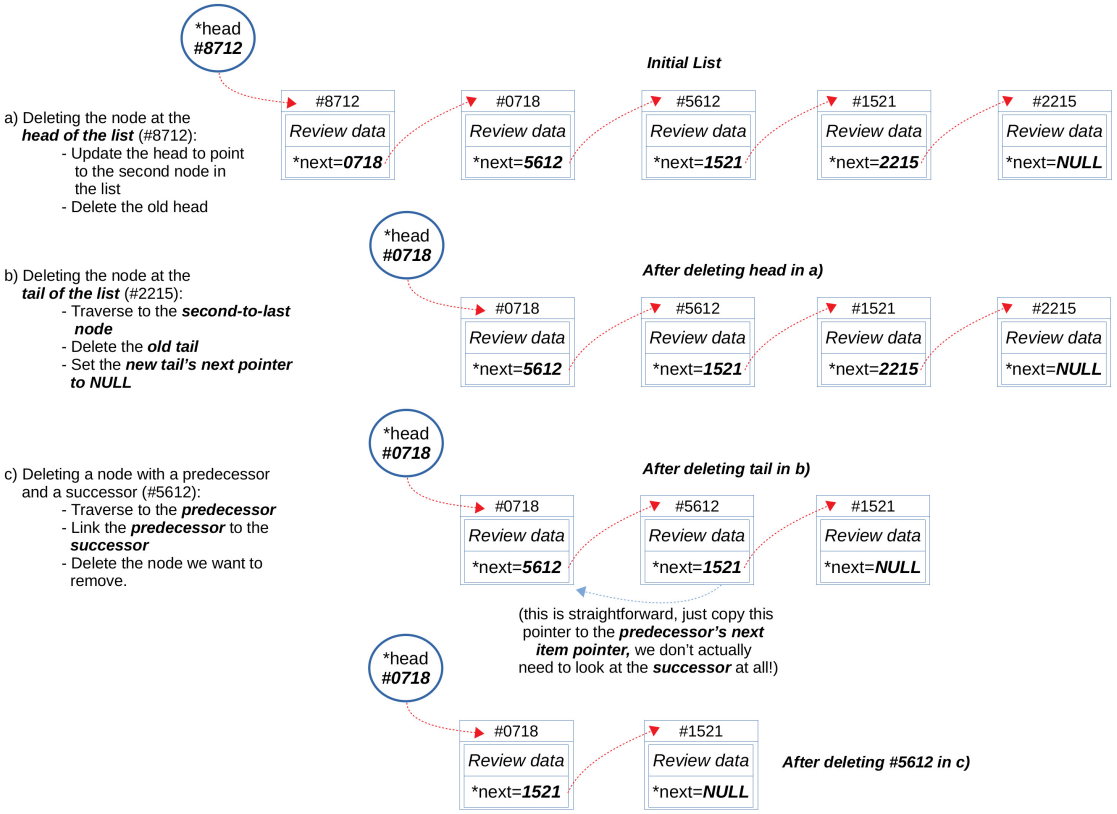

Figure 3.30: Three different cases for deleting nodes from the list, and what the list looks like after each node has been correctly deleted.

.

```
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.

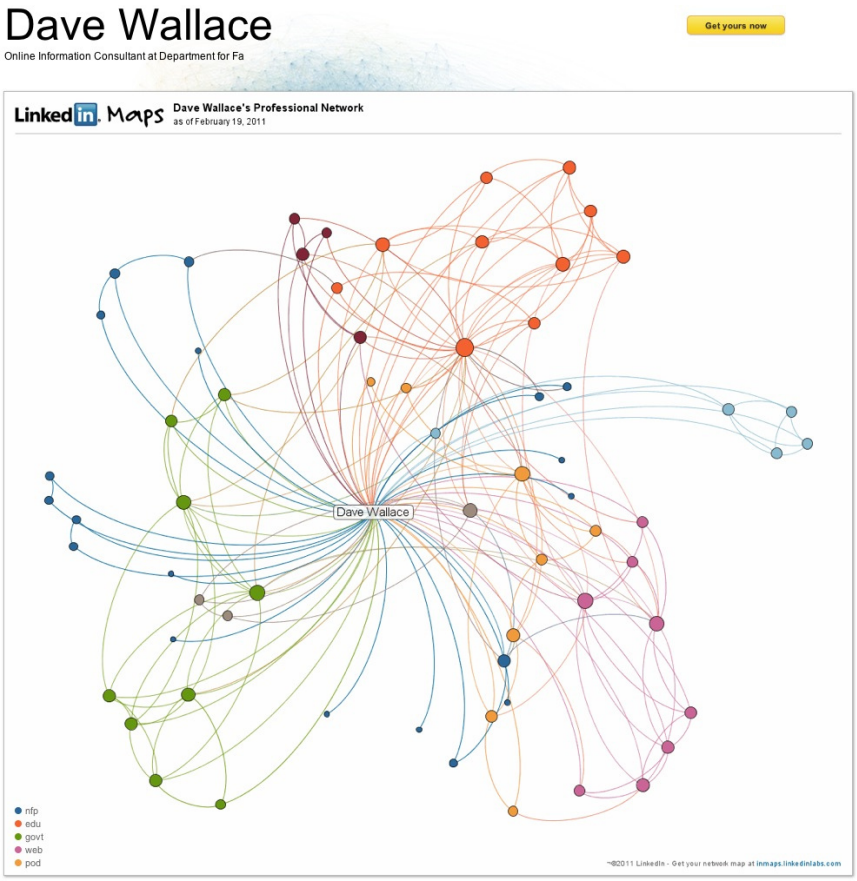

Figure 3.31: This image shows a graph for the professional connections of a single individual. Each **circle** represents an individual, and each **line** represents a professional connection between two individuals. *Image: Dave Wallace, Flickr, CC-SA 2.0*

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.