Learning Outcomes

This unit will cover the basics of how to deal with data that doesn't fit neatly into one of c's pre-defined data types. By the end of this unit, you will be able to:

- Use structs to declare your own data types
- Understand how to build and use abstract data types
- Write code to define and manipulate linked lists
- Use arrays or linked list interchangeably to implement an ADT, and understand the costs and benefits of each implementation

1 The problem with data

Now that we know enough about how to implement things in C, it's time to turn our attention to the first of the really interesting ideas we will be studying in this course.

Our goal here is to understand how to approach a general problem: how to organize, store, and access information in a computer-based system. Here are some of the things you may wish to store and organize using computers:

- Text (documents, articles, books, etc.)
- Music (in general, sound files – interviews, documentaries, sound clips, etc.)
- Pictures (in multiple formats)
- Video
- Information about people – depending on your applications

  e.g.

  Student information in ACORN
  Customer records for on-line stores
  Personal information and patient history for hospitals
  Browsing preferences (do you know what your browser knows about you?)
  etc.

- Computation results

  e.g.

  Predictions (the weather tomorrow, stock market, election results)
  Database queries (find all students enrolled in CSCA48, Lec0002)
  The shortest route to drive from one place to another (e.g. in your favorite maps application).
  Etc.

These are just a very small number of examples to give you an idea of the wide range of situations in which we will find a need for thinking carefully about. ***In general, one of the first things you have to do when solving a problem using a computer is to think carefully about:***

- What type of information we have to store
- How much of it do we need to manage (handling a few hundred pictures in your cellphone is a different problem than handling hundreds of millions on Google image)
- How the data will be accessed and used, and by whom (efficiency, storage usage, security and access control, data backup and redundancy)
- How to organize the data so it is easy to manage in a program (the actual program-level representation and manipulation of the data)

What you will learn in this part of the course is the 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 store, organize, access, and manage pretty much anything you may wish to store inside a computer.

The concepts and techniques covered here will be the foundation you need if you later wish 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 topics of Software Design, CSCB07). 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 out. Databases are fascinating, so if you're curious about how they work don't forget to check out our Intro to Databases CSCC43.

ASIDE

How much data is out there?
You may have heard the term *'big data'* mentioned out there. It is a rather uninformative term in that it really doesn't tell you how much data exactly qualifies as *'big'*, or why it should be that way. Here's a few facts (mind that some estimates are one or two yeas old!) to give you an idea of how much data is out there in different applications and for different uses.

1. The Google codebase (2016) includes approximately one billion files and has a history of approximately 35 million commits spanning Google's entire 18-year existence. The repository contains 86TBa of data, 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)
2. 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 a couple years ago when this came out!

ASIDE (CONTINUED)

3. Number of pictures uploaded to facebook each day. 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.’

   From the same site:
   https://www.omnicoreagency.com/instagram-statistics/
   We find that:
   ’More than 50 Billion photos have been uploaded to Instagram so far.’
   ’Pizza is the most Instagrammed food globally, followed by Sushi.’

4. How much new information is produced worlwide each day? According to Forbes: 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

   ’There are 2.5 quintillion bytes of data created each day at our current pace’
   ’On average, Google now processes more than 40,000 searches EVERY second (3.5 billion searches per day)!
   ’We send 16 million text messages’ (every minute!)
   ’Every minute there are 103,447,520 spam emails sent’
   Oh, and you should have a look here: http://www.everysecond.io/youtube)

Figure 1: This is what the inside of a data center looks like. Somewhere in the world a data center similar to this one is storing a copy of this very document. Photo: Global Access Point, Public Domain

IMPORTANT NOTE

A brief note on sources and whether you should believe what you read: You may have noticed that the *'facts'* listed above come from very different sources. 1) comes from a research publication, 2) and 3) are from on-line blogs/websites, and 4) from an on-line version of a magazine article.

Always remember: Before you accept a statement you read – *even in lecture notes!* you should always think about *where it came from*, and *what evidence was provided to support it.*

A good scientific journal or conference article is trustworthy: It has undergone a process of revision and review by scientists *who are not affiliated with the authors of the work they are reviewing.* This means that big mistakes or factual errors are usually caught and corrected before anything gets published.

Reputable publications are almost as trustworthy – they have a lot invested in keeping their reputation as trusted sources of information.

Blogs and websites should be taken with a grain of salt – you will notice that the websites I listed above provide references and links to the sources from which they gathered facts and figures *you should always look for this.* Any website, blog, or post that states a fact and *has no reference to a reputable source such as a scientific article, reputable publication or organization, or publicly available/verifiable survey* must be taken with a healthy dose of disbelief until *you can verify* the information independently.

You are training to become professionals in different fields – you must *always* make sure that the information you accept and incorporate into what you know of how the world works is *accurate, verifiable, and as far as you can check, correct.*

2 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 that is not available in C, and can represent a much more rich 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.

Figure 2: A 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:

Information needed for each movie:

- Title
- Year
- Director
- Studio
- Rotten Tomatoes[1] score

...etc.

There are several individual components, each of them has its own data type – there may be strings, integers, floats, and so on. How could we do this in C?

Given what we know at this point, 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:

- One array of strings for the movie titles
- One array of integers for the year
- One array of strings for the director's names
- One array of strings for the studio
- One array of floats for the Rotten Tomatoes score

Now we can write code that allows a user to fill-up these arrays with information for movies.

Question: What do you think are the advantages and disadvantages of storing the information we need in this way?

From a conceptual point of view, and also from the point of view of ease of implementing code that works with data about complex entities such as movies, it would be much better if we could *bundle* the information pertaining a *single movie* into a *single data item* that contains everything we need to store about the movie.

In C, we can define our own *composite data types*, also known as *compound data types* which are the program equivalent of a *Bento Box*: They are composed of *a set of existing data types*, each of which provides needed information about a data item, and all of which are needed to describe our data item.

Movies are fairly complex data items (if you look at the information on IMDB[1] 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 which consists of:

- Restaurant name (a string)
- Restaurant address (a string)
- Review score (an integer, let's say in 1-5)

Let's see how we can build a *Bento Box* in C that holds the information required to handle a single restaurant review:

```c
typedef struct Restaurant_Score // We are declaring a new type!
{
    char restaurant_name[1024];
    char restaurant_address[1024];
    int score;
} review; // Our new type will be named 'review'
```

The little snippet of code above works as follows:

```c
typedef struct Restaurant_Score
```

this tells the compiler we are defining a new data type (*typedef*), that the data type is composite (*struct*), that the composite's name will be *'Restaurant_Score'*, and that the new data type will be called *'Review'*. This in effect defines a new *bento box* that contains two strings and an integer. Each of these *components or parts* is called a ***field***. So our new compound data types has ***three fields***, a *restaurant name field*, a *restaurant address field*, and a *score field*.

Thereafter, we can *declare* variables of this new composite type and use them in our program!

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

#define MAX_STRING_LENGTH 1024

typedef struct Restaurant_Score     // We are declaring a new type!
{
   char restaurant_name[MAX_STRING_LENGTH];
   char restaurant_address[MAX_STRING_LENGTH];
   int score;
} Review;                           // Our new type will be named 'review'

void 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 references 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,"A nice restaurant with good food");
    strcpy(rev.restaurant_address,"Somewhere in Scarborough");

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

A lot is going on in the program above, so let's take it one step at a time.

Firstly, notice the <*include*> statements at the top:

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

The header *stdio.h* contains definitions for standard input/output functions, including the ones we will use to read strings and integers from keyboard; *stdlib.h* should be familiar to you from the previous section, it provides a number of standard functions and constant definitions (e.g. ***NULL*** is defined here), and *string.h* is the string manipulation library.

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

```
#define MAX_STRING_LENGTH 1024
```

This line defines a *constant* called *'MAX_STRING_LENGTH'*. In C, it is common to use a *'#define'* statement near the top of your program to define any constants that will be used in your code. Typically the names of these constants are all upper case. Thereafter, whenever the compiler finds *'MAX_STRING_LENGTH'* in your code, it will replace *'1024'* and use that instead.

The next chunk is our *typedef*:

```
typedef struct Restaurant_Score     // We are declaring a new type!
{
    char restaurant_name[MAX_STRING_LENGTH];
    char restaurant_address[MAX_STRING_LENGTH];
    int score;
} Review;                           // Our new type will be named 'review'
```

This will define our *bento box* with a string for the restaurant name, a string for the address, and the integer that holds the review score. At any point *after* the *typedef*, we can declare variables of our new data type just like we would any regular C types, e.g.

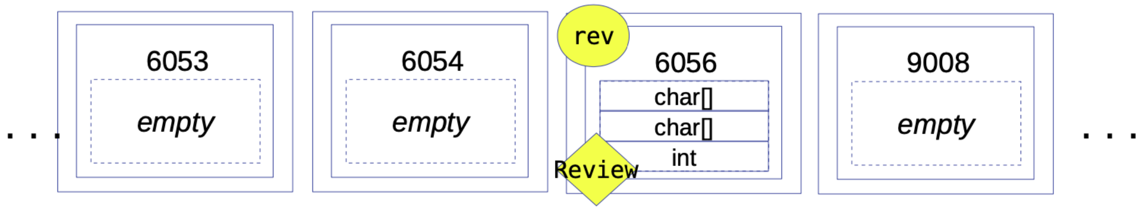
```
Review rev;
Review ten_reviews[10];
Review *rp;
```

The examples above declare (in order), one *'Review'* variable called *'rev'*. An array of 10 reviews called *'ten_reviews'*, and a pointer to a variable of type *'Review'*.

The next part of our program declares a variable of type *'Review'* called *'one_review'*.

```
Review one_review;
```

This is how it would look like in memory:



The nice thing about C, is that once a new type is declared it gets treated just like any other type. So, the *'Review'* variable *'rev'* gets a box – large enough to store what it needs, somewhere in memory just like any other variable. That box is tagged *'rev'* and it is of type *'Review'*. The box contains a collection of things now (we know there are two strings and an int in there) but otherwise it's just one more box.

When we compile and run the code above, we get:

```
...\a.exe
This review has:  name=A nice restaurant with good food, address=Somewhere in
Scarborough, score=4
```

Things to note from the example above:

- It is not difficult to define a new data type, with any mixture of types supported by C. We can use this to organize data for complex items (for example, movies, books, etc.) that our program will need to work with.

- Once we define a new data type, we can use it in our code like we would any other type. This includes using our new data type as part of an even more complex *bento box*. For example, we could define a new data type that contains items of type *’Review’*. A box is a box. So for C this is perfectly fine.

- Assigning values, or accessing data within a variable of a compound type is done via the ’.’ operator.

- You may have noted that the line in the program with the *printf()* call is split into two, and that there is a ’ćharacter at the end of the first line of this call. In C, you can split any statement into multiple lines by using a ’ćharacter at the end of one line. It allows you to write long statements across multiple lines (which makes your code easier to read) instead of one very long line.

EXERCISE

Write the definition of a compound data type that expands *’Review’* to include the name of the person who submitted the review, the phone-number for the restaurant, and a link to the restaurant’s webpage. *Think carefully about the data types you will use for this, and about how these new pieces of information might be used in the reviewing app* (which will have an effect on what data type you should use).

3 Passing Compound Types Between Functions

Like any other variable, you will find you need to pass compound data types as parameters to functions, and/or to return these compound types from functions. The way this is done is *identical* to the way you pass standard C-types between functions.

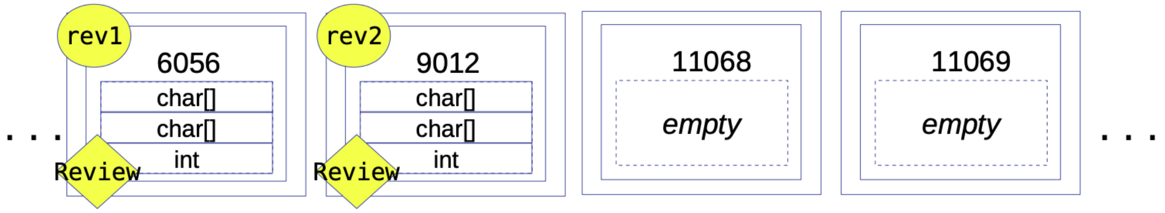
Suppose we define a function as follows:

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

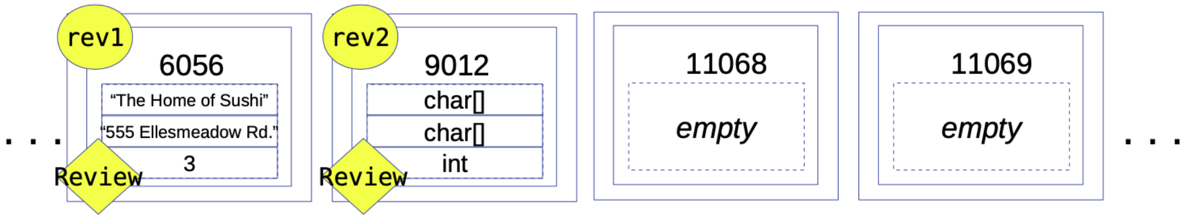
This function takes as an input parameter a variable of type *’Review’*, an *int* called *’new_score’*, and returns a result of type *’Review’*. Now, suppose that in *main()* we do something like this:

```
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);
```

Let's see what this does in memory. First, *main()* declares two variables of type *'Review'* called *'rev1'* and *'rev2'*. As expected, this will reserve boxes of appropriate size in memory, and tag them with the name of the variables:



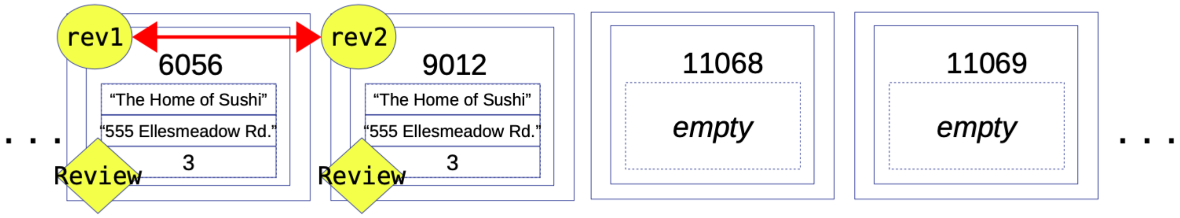
Then we assign values to the data inside *'rev1'*.



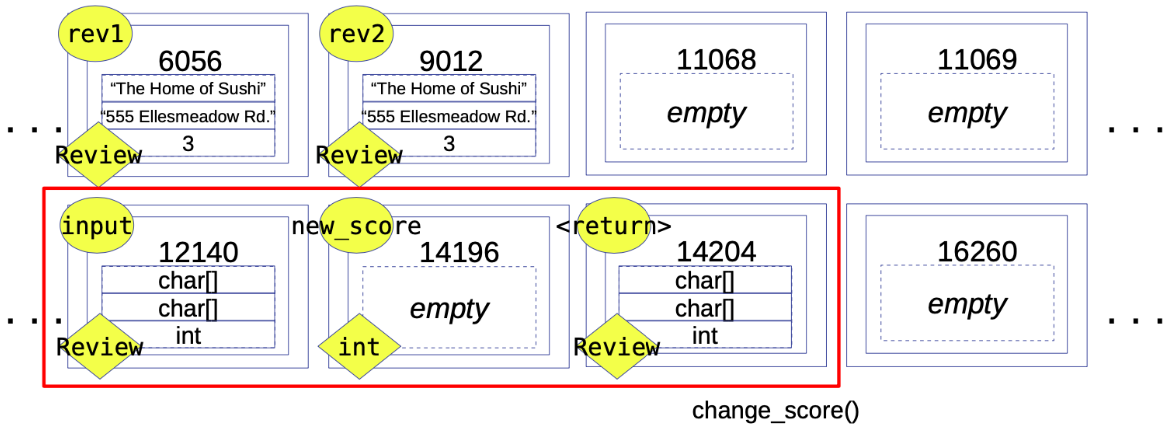
The line

rev2=rev1;

reads, *"take the contents of the box tagged 'rev1' and copy them to the box tagged 'rev2'"*. The point to be made here is that ***assignments of this type make a copy of everything inside our compound data type.***



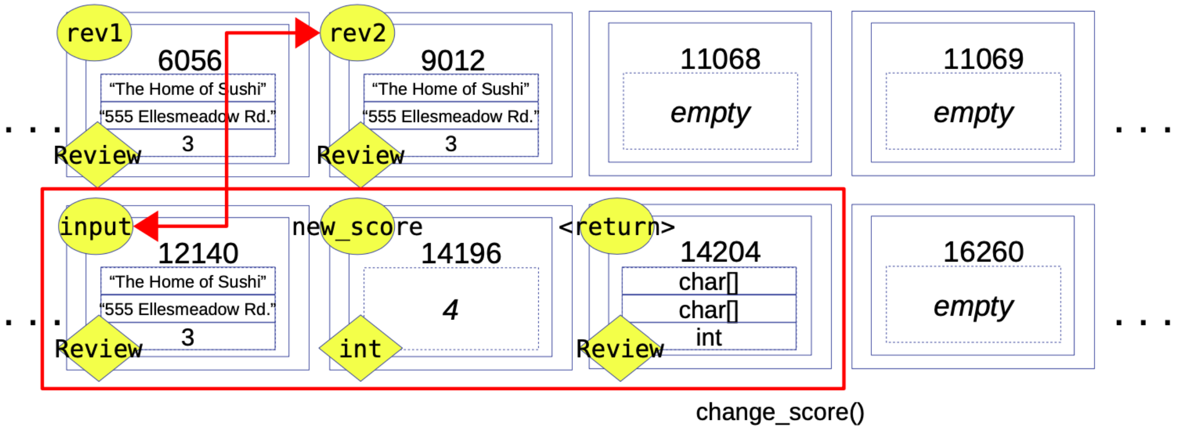
So now we have *two identical* boxes, each containing the same address, restaurant name, and score. When the call to *change_score()* happens, space is reserved for the function's input parameters and return value: The input parameters are one variable of type *'Review'* called *'input'*, an *integer* variable called *'new_score'*, and the return value of type *'Review'*.



The space reserved for parameters and return type for *change_score()* is marked by a red box above. Of course, initially it is all uninitialized. The line

`rev2=change_score(rev2,4);`

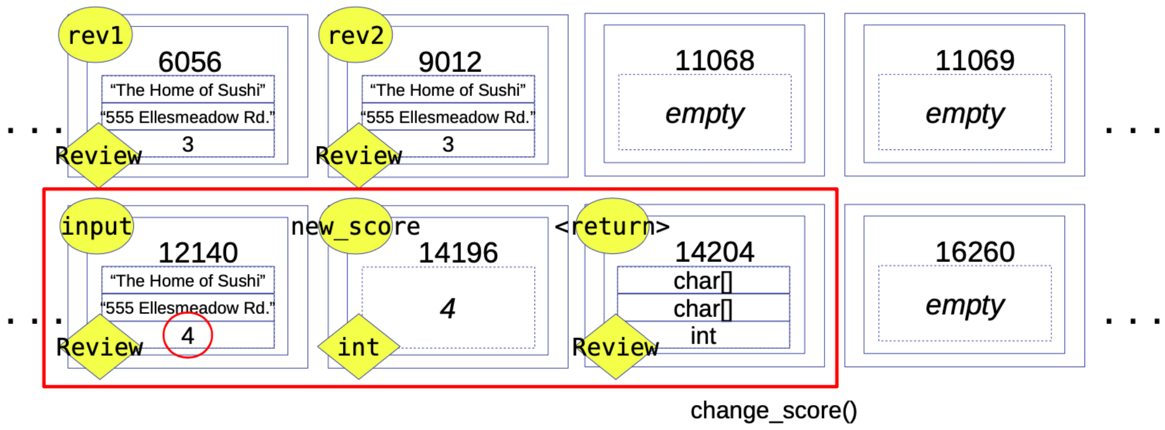
passes *'rev2'* as *'input'*, and a value of *'4'* as *'new_score'*. So, *a copy of the contents of the box tagged 'rev2' is made in the box tagged 'input'*, and stores a *'4'* in the box tagged *'new_score'*. ***Remember: input parameters are local variables with their own reserved box!***



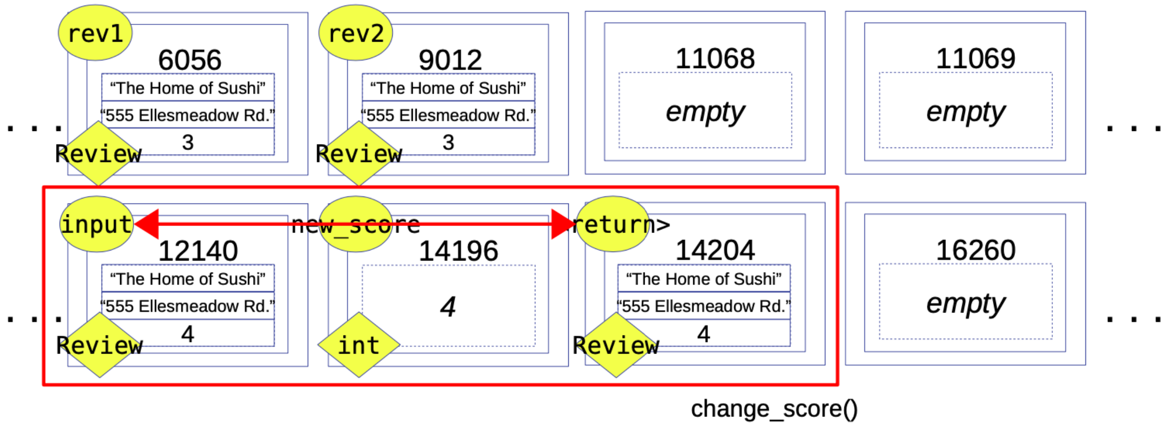
Now we have *three copies* of the same review. Each in its own box. The function *change_score()* updates the score for *'input'*.

`input.score=new_score;`

So, the score is updated in the *'input'* variable:



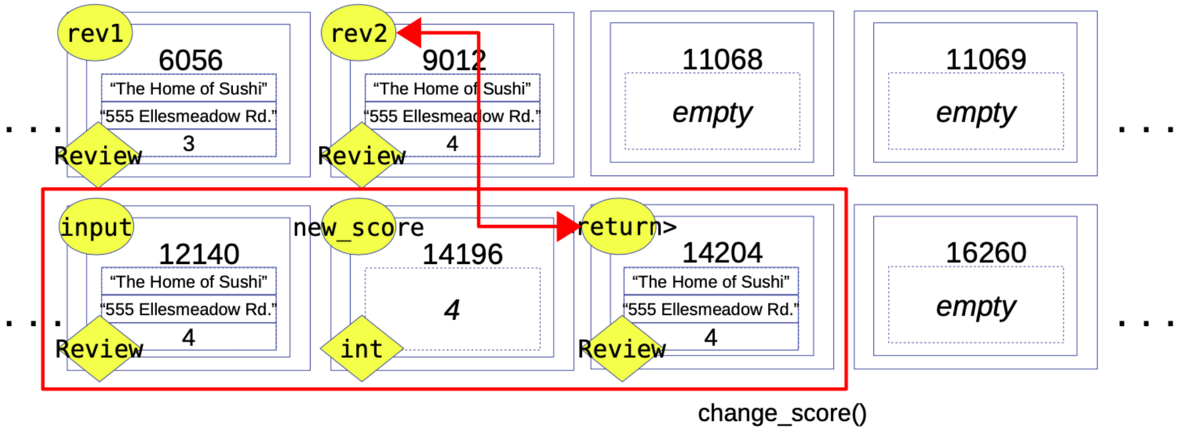
The final line in *change_score()* returns the updated *'input'* variable – this means making a copy of it to the return value box:



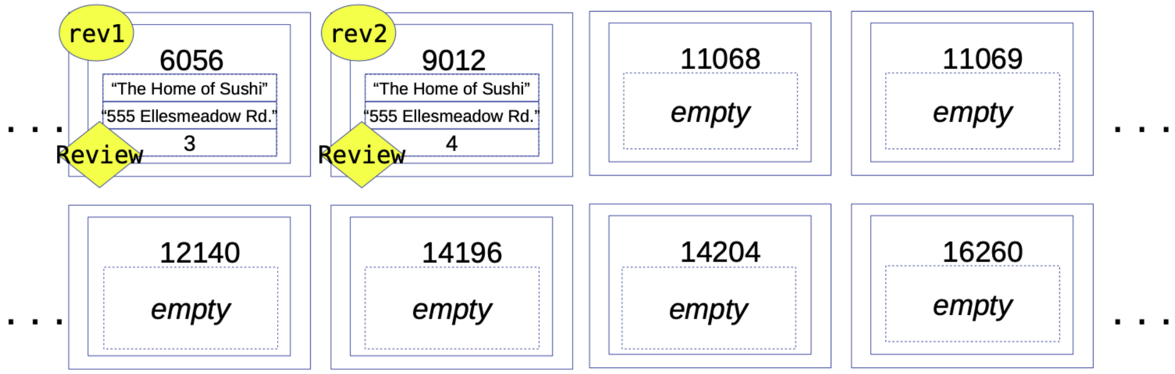
Finally,

rev2=change_score(rev2,4);

the value returned from *change_score()* is copied onto *'rev2'*. Once again, this means copying everything that is in the box labeled *'<return>'*.



Finally, the space reserved for *change_score()* is released.



What you should take from this:

- Every time a variable from a compound data type is copied, either because of an assignment, or because they are being passed as input parameters or returned from a function; a copy is made of every *field* of the compound data type. *It is never the case that individual fields of a compound type are copied, it's always the whole bento box!*

- Compound data types behave just like any other C variable. However, ***note that you can not use comparison operators on compound data types.*** E.g. the expression
  `if rev2 < rev1`
  is not valid. C has no way to compare these two compound types. If you need to compare data types you have declared, you have to write a comparison function that takes two variables of this type as parameters, compares them in a way that makes sense given what the type represents, and returns a value that indicates how they should be ordered.

- Moving data around by copying compound data types can be slow. Just like arrays, we can easily create compound data types with a fairly large memory footprint (i.e. they contain a lot of data). Copying them into and out of functions can be time consuming, as well as taking up memory.

4 Using pointers with compound data types

As we just saw, moving compound types around involves 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 compound type defined outside. Just like with arrays, the way to do this is via the use of pointers. Let's see how we use pointers to handle compound data types:

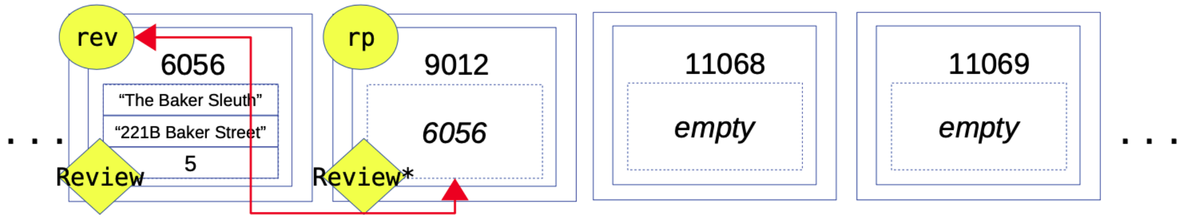
```
Review rev;
Review *rp=NULL;

strcpy(rev.restaurant_name,"The Baking Sleuth");
strcpy(rev.restaurant_address,"221B Baker Street");
rev.score=5;
rp=&rev;
rp->score=4;
```

The code above declares one variable of type *'Review'* called *'rev'*, and a pointer *'rp'* to a variable of type *'Review'*. Then we set the score for the review to a value of 5.

The next line

```
rp=&rev
```

is read *"take the address of 'rev' and store it in pointer 'rp'"*. Thereafter, *'rp'* contains the address of our review, and we can use the pointer to access and modify the information contained in that review. In memory, after the line above is executed, we would expect something like this:



Remember that to access the components of a compound data type we use the '.' operator. This works only for *variables* of that type. With pointers, we use the '->' operator:

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

the above updates the score in *'rev'* to 4 using the pointer we have for that variable. Similarly, we could change the address using the pointer:

```
strcpy(rp->restaurant_address,"221A Baker Street");
```

We will be using pointers with compound data types quite a bit, so practice accessing and modifying data with these pointers until you're comfortable with the process.

EXERCISE

Write a little program such that *main()* declares a variable of type *'Review'*, but now it calls a function *fill_out_review()* that takes a *pointer* to the review and fills out the name, address, and score of the restaurant. This function does *not have a return value since it modifies the review directly.* After the function returns, have *main()* print out the contents of the completed review.

5 Getting user input

The program we wrote in section *2* to illustrate compound types is designed only to give you an example of how to define a new data type, how to create variables of this type, and how to assign and/or access information within these variables. However, for any interesting use of our new *'Review'* type, we need to be able to get user input for the fields of the review our app will store and manage.

Let's now review how to get inputs from the user into our program.

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:

```c
#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.exe
Enter two integer numbers and one float on the same line
Separated by spaces
3 7 3.14159265
Read:  3, 7, 3.141593
```

Things to note:

- The ***signature*** (standard definition) of *main()*. Note that we declared that *main()* returns an *int*. You may wonder *why?* and *to where?* - The reason we have *main()* return a value is that we assume a program written in C may be part of a script that does multiple things and runs multiple processes on some data. The return value from a program is used by such a script to determine whether the program completed successfully or not. The convention is:
  - *main()* returns 0 (zero) if it completed successfully
  - *main()* returns a non-zero value if there was an error during execution
- The *formatting string* for *scanf()* specified 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.
- 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 further input.

- 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 read from the terminal:

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

  The line above reads: *scan from the terminal one int, one int, and one float, and store them at the address of 'x', the address of 'y', and the address of 'pi'.* The data types for *'x', 'y',* and *'pi'* are *int, int,* and *float* which matches the formatting string we provided to *scanf()*.

IMPORTANT NOTE

Always make sure the formatting string and the variable types used to store the values read from the terminal match. The *scanf()* function will not warn you if the types don't match, and you will end up with *junk* values in your variables. Also remember that *scanf()* can not protect you from the user typing junk, or providing values that don't match what is expected

To make sure you remember this, see what happens when we run the same program, but the user is being unhelpful:

```
...\a.exe
Enter two integer numbers and one float on the same line
Separated by spaces
hafs kjas 5
Read:  32767, 0, 0.000000
```

That clearly makes no sense. ***You should always check that the user input is sane 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 again to input a valid score.

EXERCISE

Write a little program that declares an *int array* with 10 entries, 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!)

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 indicate 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 String).

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

```
#include<stdio.h>

```

```c
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);`

The first input parameter is the name of the string variable where to store what is read. The second parameter specifies the *maximum number of characters to read and must be less than, or equal to the size of the character array for our string* (***note:*** *fgets()* will read one byte less than the specified maximum number, because it needs to add the *end-of-string* delimiter *’\0’* to the string, so in the case above it will read at most 1023 characters from the terminal). The final parameter specifies *where to read from*. The function *fgets()* can be used to read from different sources of data, including files, the name *’stdin’* corresponds to the *standard input*, which is the terminal. We will see later how to use *fgets()* to read strings from other sources.

Compiling and running the program above results in:

```
...\a.exe
Please type one string
Hello World!
The input string is:  Hello World!
```

IMPORTANT NOTE

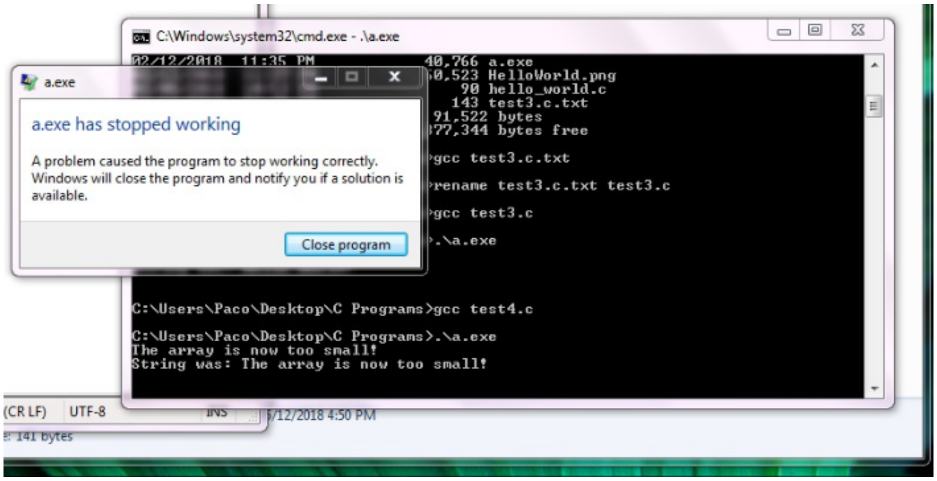
Be careful your string arrays are large enough to contain the information you will need, and use *fgets()* carefully. Trying to store a string that is too long within a small array will crash your program.

See what happens when we change the length of the character array to 10, but forget to change the maximum number of characters to be read from the terminal:

```c
#include<stdio.h>

int main()
{
    char my_string[10]; // This is now too small!

    printf("Please type one string\n");

    fgets(my_string, 1024, stdin); // fgets() thinks we have 1024
```

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

Let's compile and run the above program:



C:\Windows\system32\cmd.exe - .\a.exe

a.exe

a.exe has stopped working

A problem caused the program to stop working correctly. Windows will close the program and notify you if a solution is available.

```
02/12/2018  11:35 PM            40,766 a.exe
                                ,0,523 HelloWorld.png
                                    90 hello_world.c
                                   143 test3.c.txt
                                91,522 bytes
                               77,344 bytes free

                               >gcc test3.c.txt

                               >rename test3.c.txt test3.c

                               >gcc test3.c

                               >.\a.exe


C:\Users\Paco\Desktop\C Programs>gcc test4.c

C:\Users\Paco\Desktop\C Programs>.\a.exe
The array is now too small!
String was: The array is now too small!
```

So just be careful with your strings and all will be well!

EXERCISE

Modify the program we wrote to declare and initialize a single restaurant review so that this time it asks the user for the restaurant's name, address, and score, and then prints the resulting information.

6 Handling a realistic amount of data

At this point we know how to create custom boxes to store information, it's time to turn our attention to one of the *fundamental ideas* this course 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 data type 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
- 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 way we know of storing multiple data items of a given 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, and they would be easily accessible to our program.

Question: Can you see a possible problem with doing things this way?

So we chose a very small size for our array. We will likely run out of space to store reviews well before we have information about even a small fraction of the restaurants out there. So, we think for a bit and decide to be clever, at the top of our program we have:

```
#define MAX_REVIEWS 100000
```

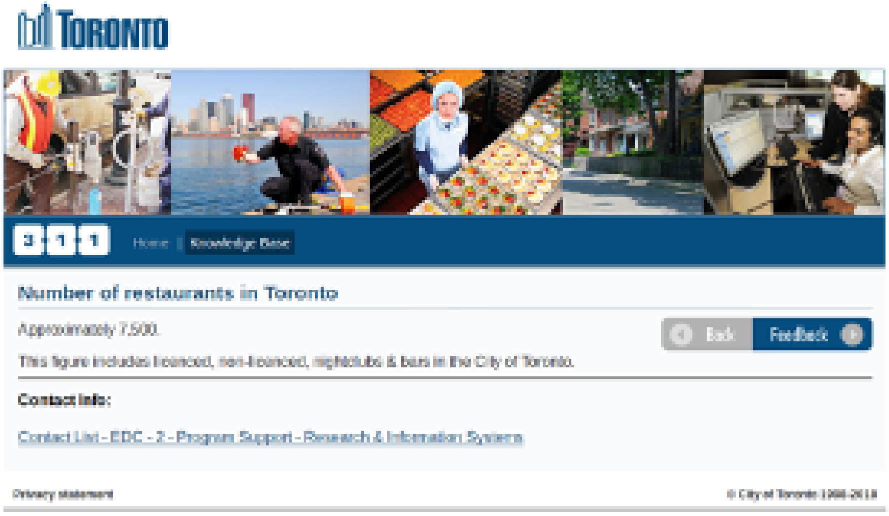
And then at the place where we declare our array we have:

```
Review all_reviews[MAX_REVIEWS];
```

Now, this is *better* in that we are unlikely to run out of space where to store the reviews for our city. And we may think we have solved all our problems. However, this solution turns out to have problems of its own.

Question: What are the disadvantages of defining the array as we did just above?

- Suppose there are only 7500 restaurants in Toronto (actually this is accurate as of Dec. 2018).



Number of restaurants in Toronto

Approximately 7,500.

This figure includes licenced, non-licenced, nightclubs & bars in the City of Toronto.

Contact Info:

Contact List - EDC - 2 - Program Support - Research & Information Systems

Figure 3: Number of restaurants in Toronto, from the City of Toronto web portal: https://www.toronto.ca/311/knowledgebase/kb/docs/articles/economic-development-and-culture/program-support/number-of-restaurants-in-toronto.html

Does it now look like a good idea to have an array with 100,000 entries?

- Suppose we want to expand our app to add a 'City' field to the reviews, and allow reviews from anywhere in the world. What can we expect will happen with our array at some point?

- Have we solved our storage problem?

What you should take from this:

- Arrays are wonderfully useful when you have a known amount of data to work with, and need a simple, easy to use way to store and manage this data. They are commonly used in data processing applications to represent and manipulate numeric data.
- They have the following limitations:
  - Their size is fixed. They can't grow or shrink to accommodate your data needs
  - Changing the array size would involve modifying your code and re-compiling it
  - If we choose a very large array size to avoid running out of space we are likely to end up with a lot of unused space most of the time (this is bad because that space can't be used by other programs – or our own). We are wasting valuable space.
- Because of these limitations, they are really not the right tool for information storage and retrieval systems – you would not implement a database using arrays for your data.

Here's the problem we would like to solve:

We need to develop a way to:

- Provide way to store, keep organized, and update a *collection* of *items* that contain the data our program will manage (e.g. we would like to have a *collection* of reviews for 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 to 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 *extendable.*
- Our solution should enable us to *search for, access, modify,* and *delete* any individual data *items* in our *collection.*

7 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 in disk 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.

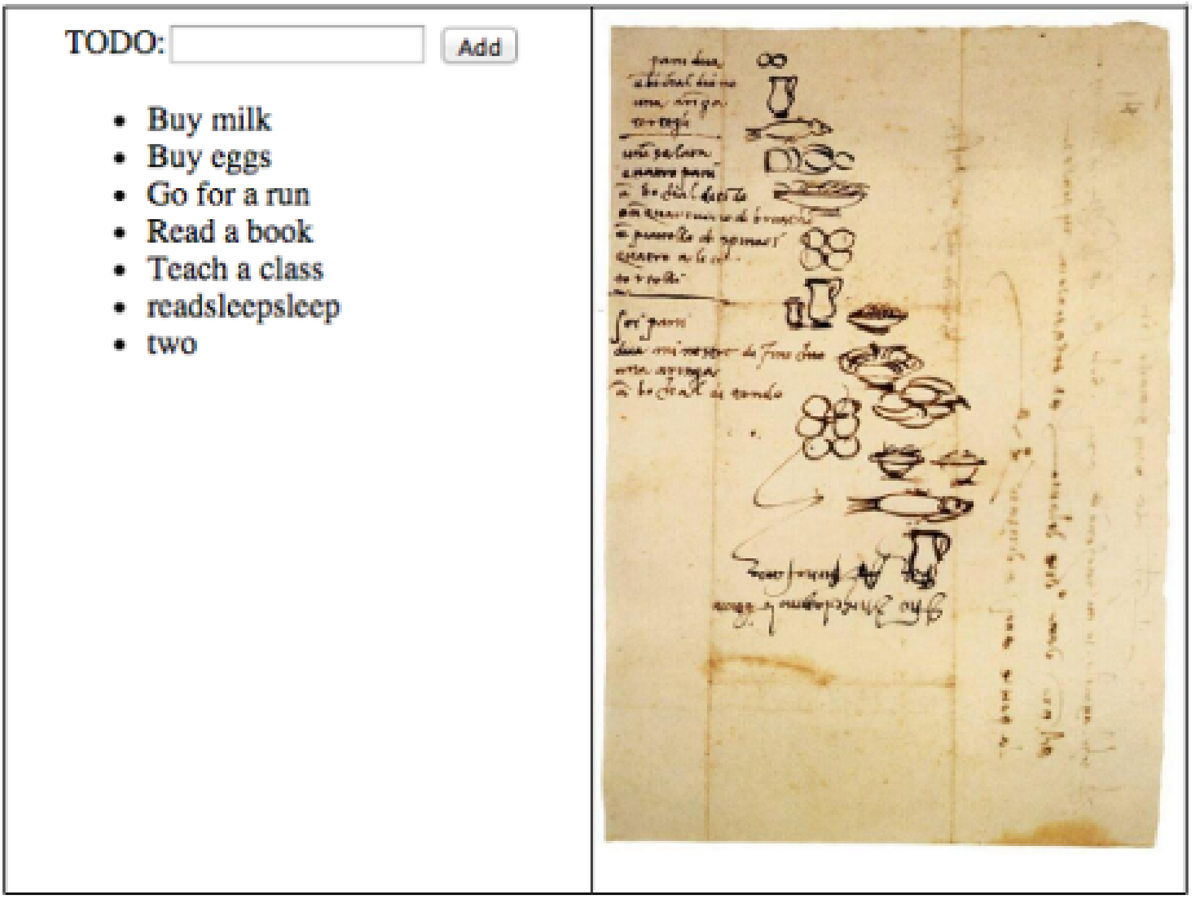
Let's have a look now at what is possibly the simplest *container* that:

- Allows us to keep a *collection* of data items.
- The collection size can grow or shrink over time.
- Memory for items is reserved *on demand*, only when needed to add a new item to the *collection.*
- It provides a means *finding specific items*, as well as *adding, deleting, or modifying existing ones.*

The *container* we are talking about is called a *List*, and it has the following properties:

- It provides a means for storing a *collection* of data items of a given type.
- 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 definition 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*, it is still a list!



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

Figure 4: 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

To make the point perfectly clear, in the picture above 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 having a collection of items, sequentially ordered, and so they both are examples of *a list.*

Let's take this concept a bit closer to a description of a list we could actually implement with a computer.

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:

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

The *searching* operation is needed so we can find and modify 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. We should also check whether a restaurant for which we are entering a review is already included in our collection.

There are variations on the definition above. You 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 you will find on pretty much any list you will find or use while programming.*

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. *Implementations* of the List ADT could be completely different from one another, and yet, anyone who knows what the List ADT includes will know what to expect: *a collection of data items, sequentially ordered, that supports declaring a new list, adding and deleting items, and search.*

This is very important 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.

ASIDE

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

If you are continuing in CS, you will learn a lot more about ***ADTs*** in your second year *Software Design (CSCB07)*, and *Design and Analysis of Data Structures (CSCB63)* courses. So ***do not forget: ADTs define implementation-independent means for storing, accessing, modifying, and organizing data.***

How do we use an ADT?

When we are solving a problem, one of the key early decisions we have to make is *how are we going to store and organize the data our program needs to work with.*

This means you have to consider the different *containers* you know of, what properties they have, what operations they provide, and how *efficient* they are; then choose the one that is most suitable to your particular problem.

For instance, at this point you know how to use *arrays*, and now you know about the *List ADT*. When deciding how to organize data for the restaurant review app, you would consider both of these, figure out which is best suited for this particular task, and then use it. We have already seen that in this case the array solution has many disadvantages and limitations, and it looks like the *List ADT* provides the properties we need and the operations our app will require in order to work, so we would choose the *List ADT*.

Once you have selected the ADT you want to use to solve a problem, you can continue working on the solution *assuming you can count on an implementation of the ADT that offers all the required operations and storage characteristics.*

ASIDE

Over time, you will learn many different ***ADTs*** each with their own properties, advantages, and limitations, and each of which is better suited to specific types of problems. The issue of *efficiency* will become very important (and in fact we will take a look at it soon!). *Your task is to understand how these ADTs work, when it's best to use them, and what advantages/limitations they bring so you can choose the best way to organize data for any problem you will need to solve.*

For now, let's assume we have decided to store our restaurant review using a *List ADT*. ***We will now use the specification of the List ADT to come up with an implementation that we can use in our program.***

8 Linked Lists

Perhaps the simplest implementation of the *List ADT* that has all the properties and operations defined by 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 *Zurich* 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).

You go out 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 hung from your neck. 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

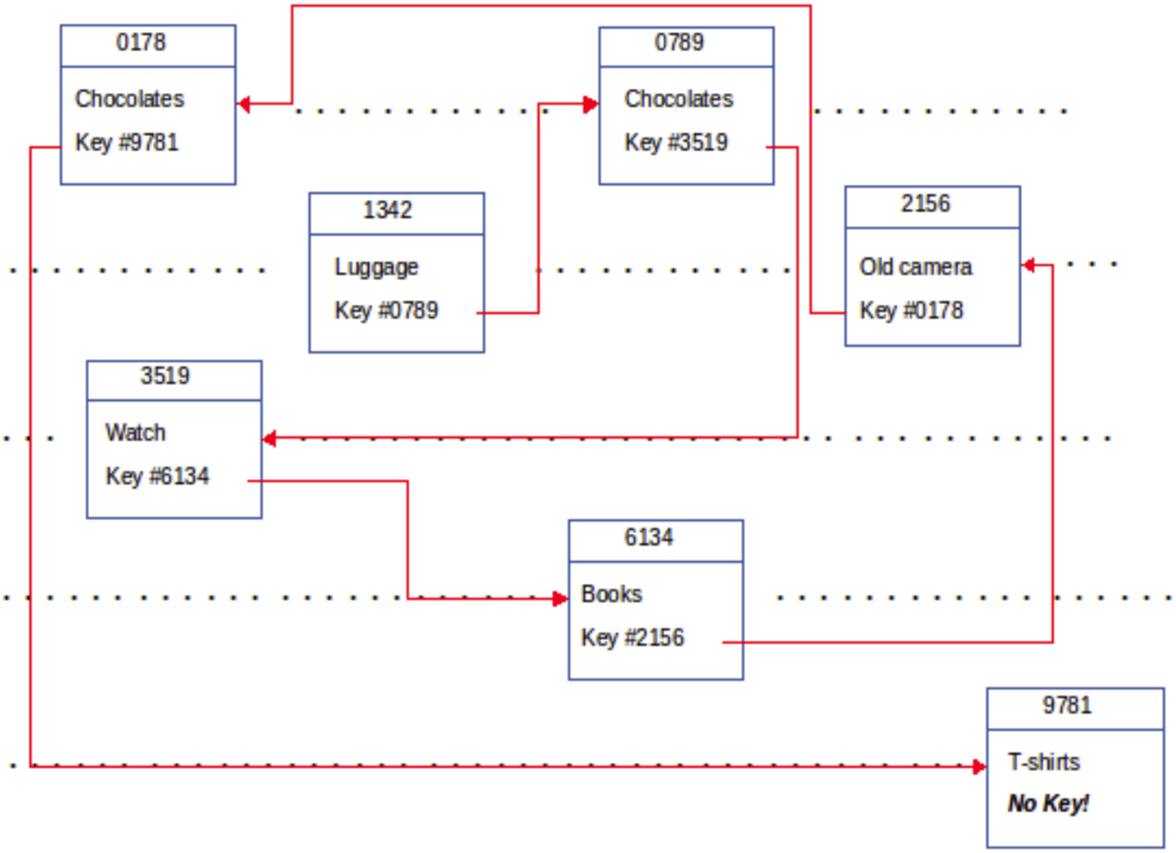
Now, go to locker #0178 and ***put inside the key for locker #9781***
Go next to locker #2156 and ***store there the key for locker #0178***
Head to locker #6134 and ***leave there the key for locker #2156***
Walk to locker #3519 and ***put there the key for locker #6134***
Move to locker #0789 and ***leave there the key for locker #3519***
Go to the first locker #1342 and ***store there the key for locker #0789***
Now you can walk outside again, carrying only the key for locker #1342!

What you have accomplished here is to create 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.***

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.

Let's look at an illustration of the locker room to see what's happening



Important things to note in the diagram above:

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

Questions:

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

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, looking in *each locker in sequence*, until we find the item we are looking for, or we reach the end of the list (in which case the item is not there!)

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 tell, that took some work.

IMPORTANT NOTE

Remember: *Whenever you 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.*

EXERCISE

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 your collection of lockers.

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 our collection. This is the crucial step for making sure our linked list remains connected.
- How to link the new locker to the list depends on ***where*** in the list we want to ***insert*** it.

Example: 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 (#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, the new ***head***.
  - The ***current head*** will become the ***second locker*** in the list.
  - We have the ***key for the current head*** (#1342) with us.

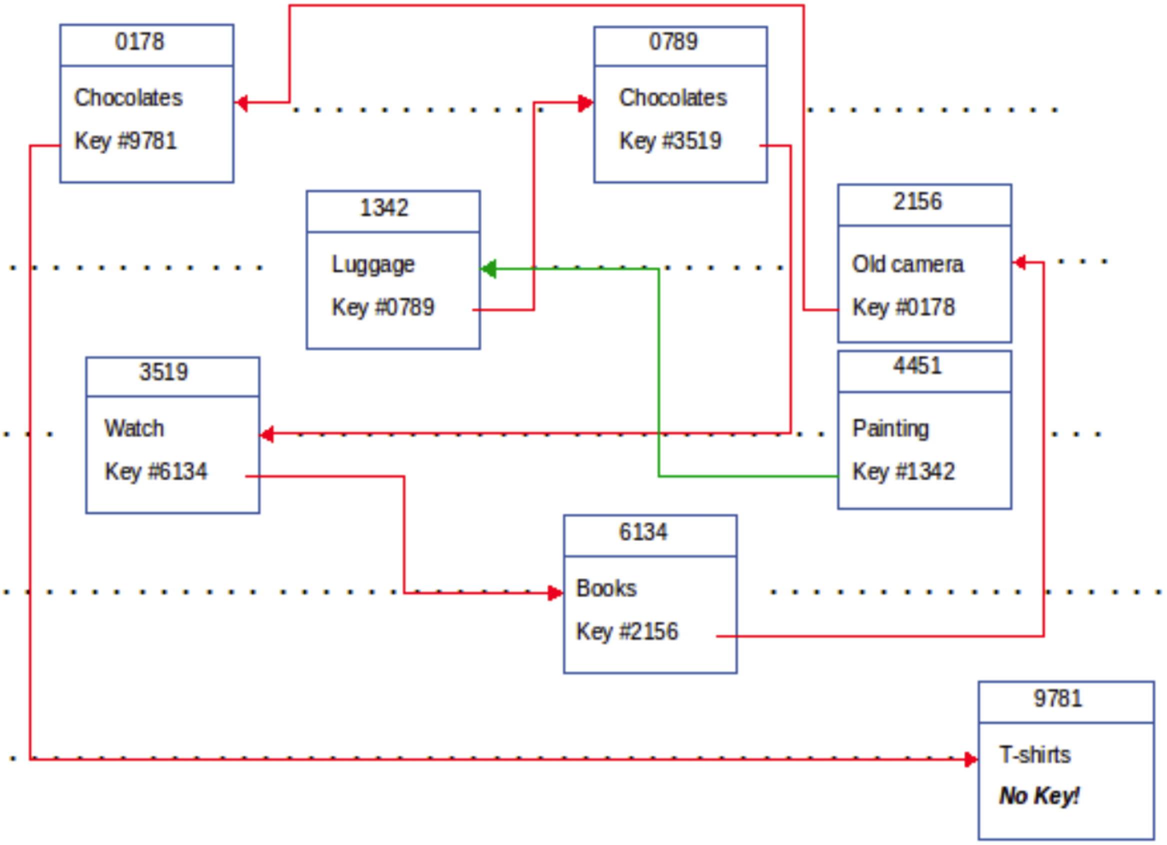
    So the process is

    Store the ***key for the current head*** (#1342) in the ***new locker*** (#4451)
    Locker #4451 is now ***the head of the list*** so we keep the key with us.
    Done!

Carrying out the above steps leaves our collection looking as shown in the figure below (the new link added from the ***new head of the linked list*** to ***the old head of the linked list*** is shown in green).

The same process would allow us to add any number of items at the ***head*** of the list. The list will grow from the front end.

Exercise: 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 a the head of the list***.



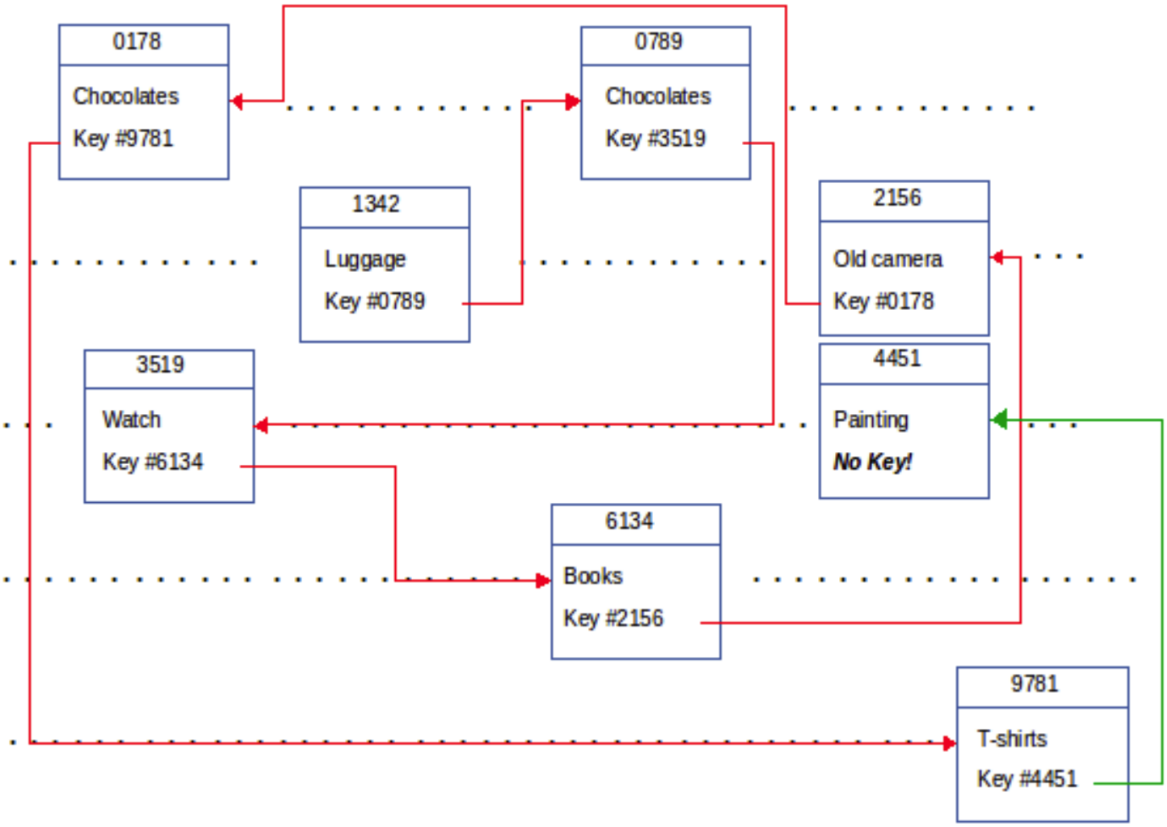
Inserting a new item at the tail of the list

Inserting at the ***head*** 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.

Suppose we wanted to add the new locker with our painting (#4451) not at the ***head*** but at the ***tail*** of the list. We would have to:

- Get the key for our locker (#4451).
- Store the painting in the locker, note that this locker will contain ***no key*** since it will be the last one in the list.
- ***Traverse*** the linked list until we reach the last locker (recognized because it has ***no key*** in it) in the case above, that would be locker #9781. This locker is the ***current tail*** of the list.
- Put the key to the new locker inside the ***current tail*** of the list.

The newly added item becomes the ***tail of the list***. If we had chosen to add the new locker to the ***tail*** instead of the ***head*** of the list, our list would look as shown below:



IMPORTANT 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 see shortly applications where adding items at the end of a list makes perfect sense.

Exercise: 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? (locker number:)
- What is the ***tail*** of the list? (locker number:)

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.

Suppose we wanted to have 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 have the key for #4451 (we got it when we reserved the locker)
  - Locker #6134 contains the key for locker #2156

    So:

    - 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)
    - We store the key for locker #2156 (old camera) in locker #4451 (painting)

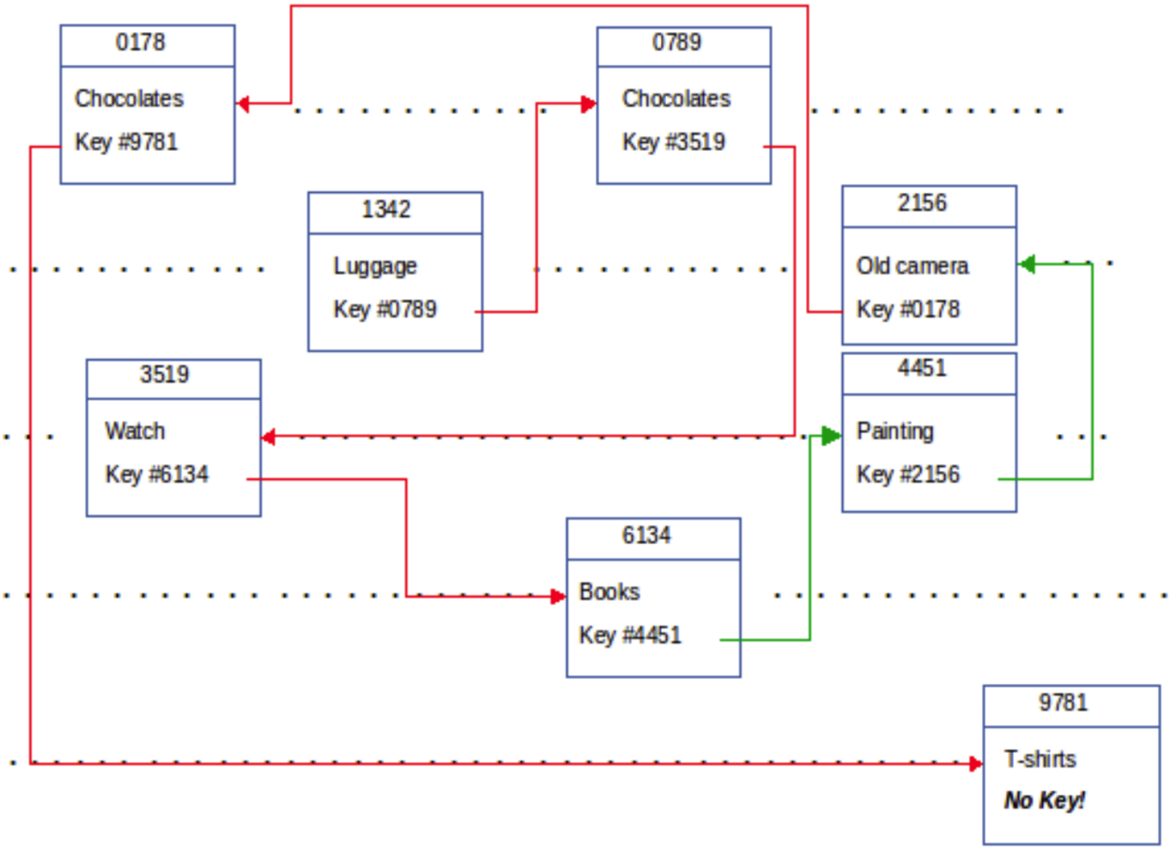
    That's it. Notice that we don't have to do anything with the contents of locker #2156, as far as that locker is concerned, nothing happened!

The 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!

If we carried out the steps above to add the painting to our collection, our ***linked list*** would look as shown below (the updated links are shown in green).

EXERCISE

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 show what the resulting list looks like.



Things 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***.

There is one final operation that we can perform on a linked list that we should look at, and then we can go ahead and write an implementation of a working linked list in C.

Removing (deleting) items from our collection

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, ***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 ....)

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

- Go to the ***predecessor*** of the item we are removing (the item immediately before)
- ***Replace the link in the predecessor*** with the ***link to the successor*** of the item we're removing (this link is stored with the item we're removing!)

In the case above, we need to take the key from #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 ....)

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.*

ASIDE

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.*

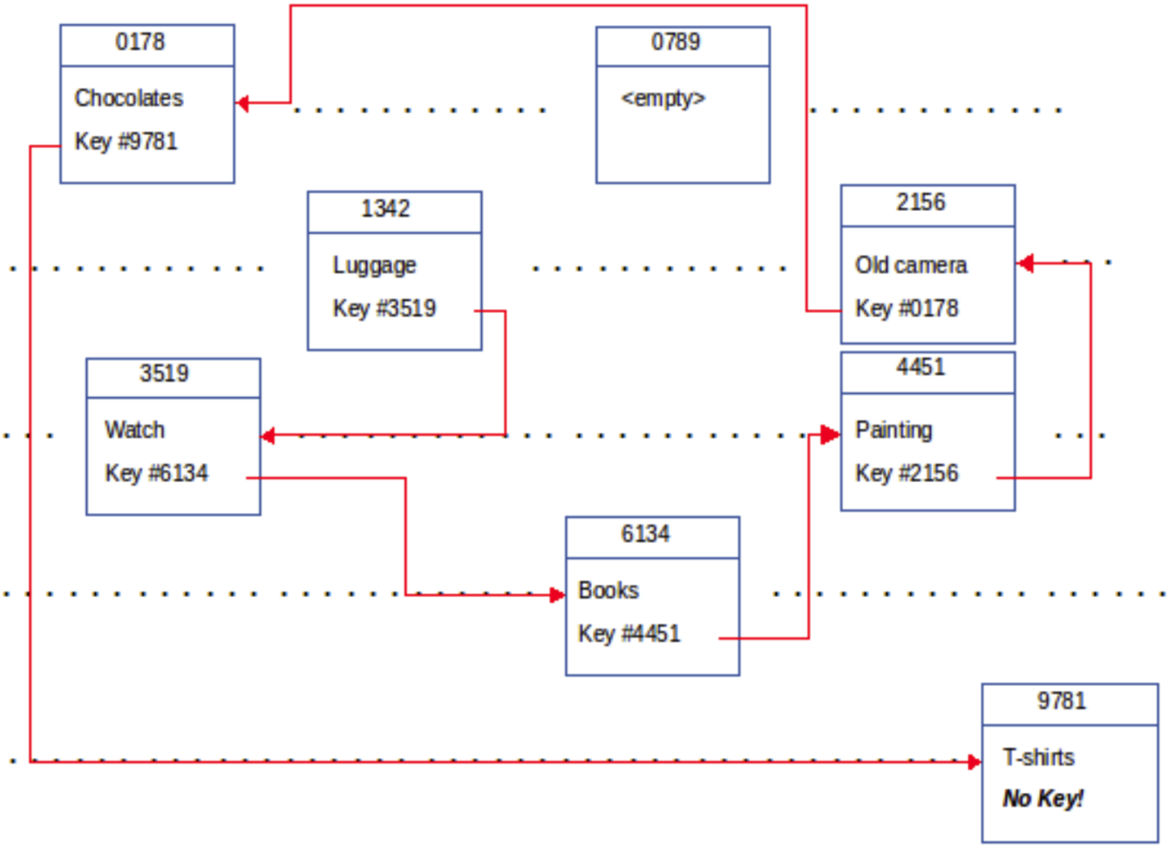
If we removed the locker with the chocolates our list would look as shown below.

Questions:

Does the same process work if we are ***removing the tail of the linked list?***
Does the same process work if we are ***removing the head of the linked list?***

ASIDE

You got through the entire example of implementing a linked list with lockers! You may be wondering why we did it this way – without any actual code. The reason is that ***the same process applies to linked lists independent of what language you're programming with, or what items you're 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. ***This is precisely the kind of understanding you should always try to achieve.*** 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.



9 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*.

ASIDE

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 the implementation 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. ***Both the data structure and the ADT describe the same way of organizing the data, and the operations that can be performed on that data.***

What we are about to do is create a linked list data structure in C. This involves the following steps:

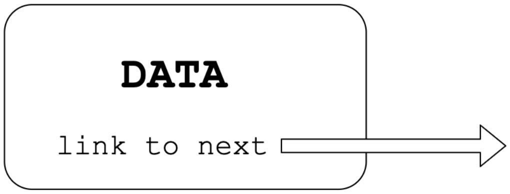
- Setting up a new data type to store ***one item*** in the list. Each individual item is usually called a ***node*** in the list.

- Setting up a pointer to keep track of ***the head of the list***.
- Writing a function to create a ***new empty node*** on demand.
- Writing a function to ***insert*** new nodes onto the list.
- Writing a function to ***search*** for a specific ***item***.
- Writing a function to ***remove*** items from the list.

It seems like a bit of work, but as we shall see the process is identical independently of what the linked list contains, so *once you know how to do this for items of one type, you can do it for items of any other type!*

Let's start with the ***node*** data type.

The general structure of a node in a linked list is



As you can see the ***node*** is just a box, and this box has ***2 parts***. The first part, the ***DATA*** part consists of the information you actually want to store in the linked list. It can be:

- A simple data type, such as *int, float, or string*
- A compound data type, such as the *Review* data type we defined earlier
- A combination of multiple data types (in effect, it can define a new compound data type)
- A *pointer to information stored elsewhere*

The point to make here is that the way we build and work with the linked list is the same regardless of what data is stored in it.

The second component in the ***node*** is the ***link*** to the next node in the list (remember the key to the next locker in our long example above!). In C, this is just a ***pointer variable*** that contains the address of the next node in our list.

We have to use pointers because as we have learned

- We don't know where in memory a new ***node*** will be placed.
- We will request space for ***nodes*** on demand, and will request as many as we need but no more.

- 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 define a linked list node for a simple data type.

Example: Define 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 data types. A linked list ***node*** is a new data type and is defined in exactly the same way. The first line

typedef struct int_list_node

tells the compiler we're 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*). The final 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 by doing

int_node a; // A variable of type int_node
int_node *head; // A pointer to an int_node

Let's see how we'd use our new *data type* in a little program!

```
#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:

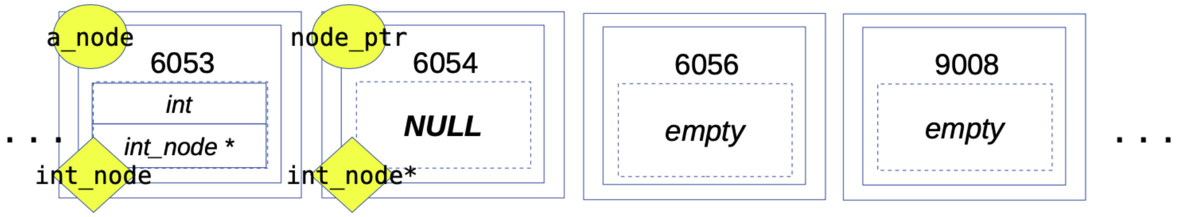
```
...\a.exe
The value contained in the node is 17
```

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, this will reserve one box of the right size to hold an *int_node*, and one box for a *pointer* as shown below



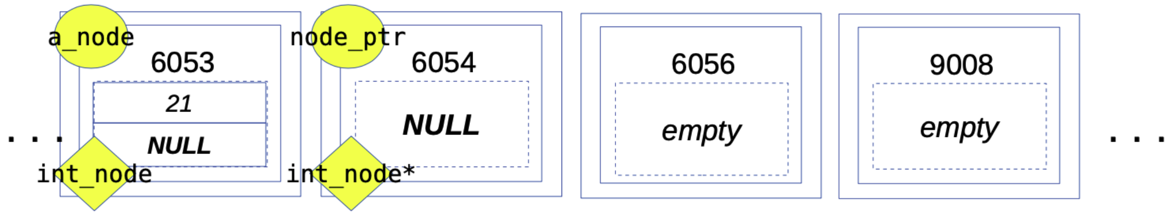
Note that:

- 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 ***NULL*** indicating it's not pointing to anything.

Next, the program *fills-in* the data in *'a_node'*.

```
a_node.stored_integer=21;
a_node.next=NULL;
```

In memory now we have something like this:



Note that:

- We set the value of a_node.next to ***NULL*** to indicate this node is currently ***not linked to anything***.

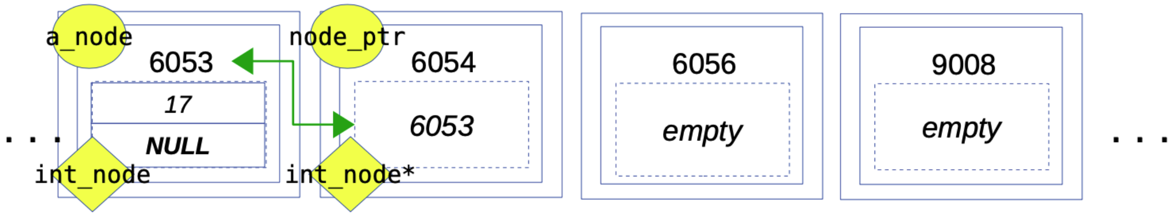
IMPORTANT NOTE

Always make sure the pointer(s) inside a newly created list node are set to NULL, otherwise, as you know, the actual memory reserved for the node will contain *junk* and your code will take this junk to be an actual pointer. This will create a hard-to-fix bug in your code!

Next we get a pointer to our newly created node, use it to change the value of the data in the node, and 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);
```

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 parts 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. In memory we now have



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 to it 'node_ptr'.

The example above is to show you how we define a ***node*** data type, and how we can declare variables and pointers of this type, and 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!

Next, we will 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 you need it*). We will see that the only way to deal with such data is by using pointers. At this point 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.

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 build 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 a 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 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.*

Creating nodes on-demand

Since we must be able to create ***nodes on-demand***, we need to write a little function that will

- Reserve space for a new *'Review_Node'*.
- Initialize the contents of the newly reserved node.
- 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; // Pointer to the new node

    new_review=(Review_Node *)calloc(1, sizeof(Review_Node));

    // 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;
}
```

Let’s look at the code above in detail – ***it’s important because it shows how you will be creating nodes on-demand for any linked list you will write in C (as well as for many other data structures).***

First, the function declaration:

```
Review_Node *new_Review_Node(void)
```

This states that the function called *’new_Review_Node’* has *no input parameters*, 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()* function that returns the size in bytes of any data type known to our program!

What does *calloc()* do?

- It finds an available place in memory that has the requested capacity

- It reserves the exact amount of space we asked for
- It wipes-out the contents of that memory space with zeros
- It returns a pointer to our reserved chunk of memory

The function *calloc()* returns a pointer *without any attached data type*, 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 to a new_review*
- We used *calloc()* to reserve memory space for the *new_review node*. It gives us a *pointer* to our *newly reserved node.*
- We store that pointer so we know where our node is

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 *values that show the node has not been updated with actual data.*

This is an important step and helps us avoid bugs caused by accessing 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 you will not mistake *junk* left over in memory by something else for a *valid pointer* to a node in 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.

```c
#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
```

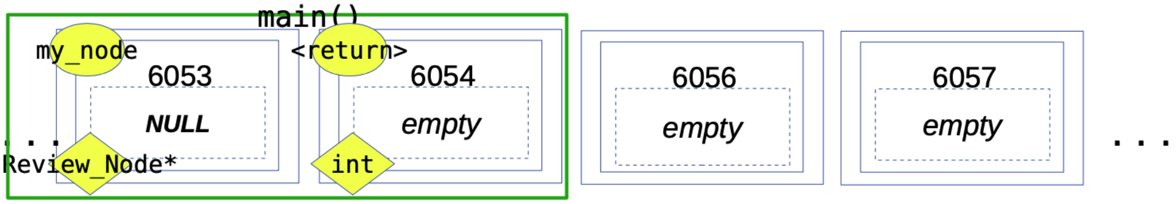
```c
{
    Review rev;
    struct Review_List_Node *next;
} Review_Node;

Review_Node *new_Review_Node(void)
{
    Review_Node *new_review=NULL;
    new_review=(Review_Node *)calloc(1, sizeof(Review_Node));

    // 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;
}

int main()
{
    Review_Node *my_node=NULL;

    my_node=new_Review_Node();

    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
}
```

Compiling and running the code above produces:

```
...\a.exe
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 all the information that makes up a *'Review_Node'*.



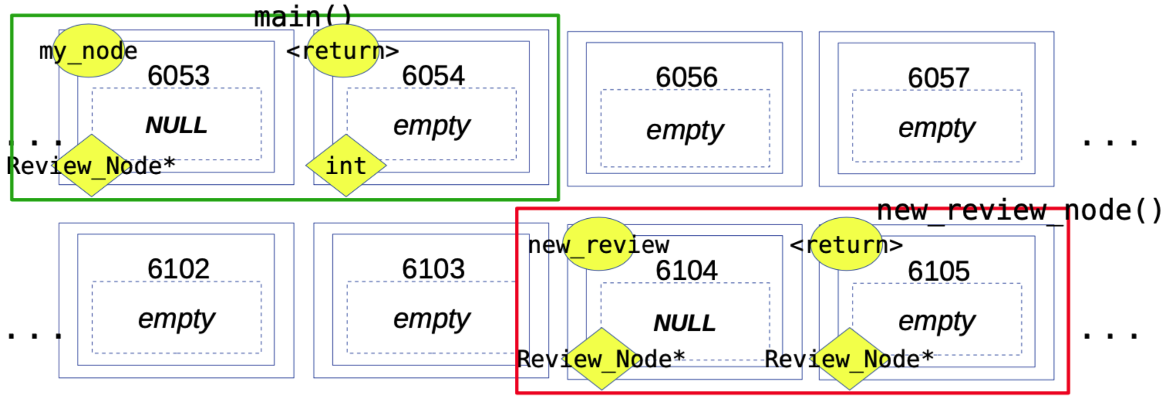
Things to note:

- The pointer *'my_node'* is the only variable declared in *main()*
- It is ***not*** a *'Review_Node'*
- It's initialized to ***NULL*** to indicate it's unassigned
- Let's not forget about main()'s return value!

Next we have a call to *new_Review_Node()*,

`my_node=new_Review_Node();`

this function 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:



Things to note:

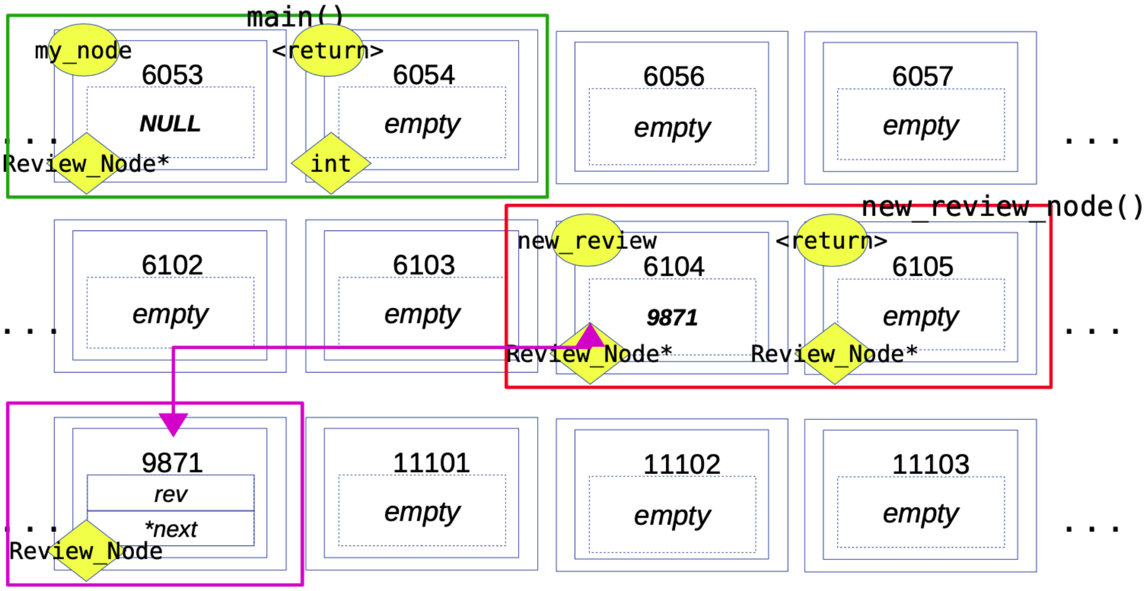
- All the space reserved for *main()* is inside the green box
- All the space reserved for *new_Review_Node()* is in the red box

- Neither of these functions has declared a *Review_Node* variable!

Within *new_Review_Node()*, memory is reserved for the new node, and we get a pointer to the newly reserved space

`new_review=(Review_Node *)calloc(1, sizeof(Review_Node));`

In memory, the result would look like this:

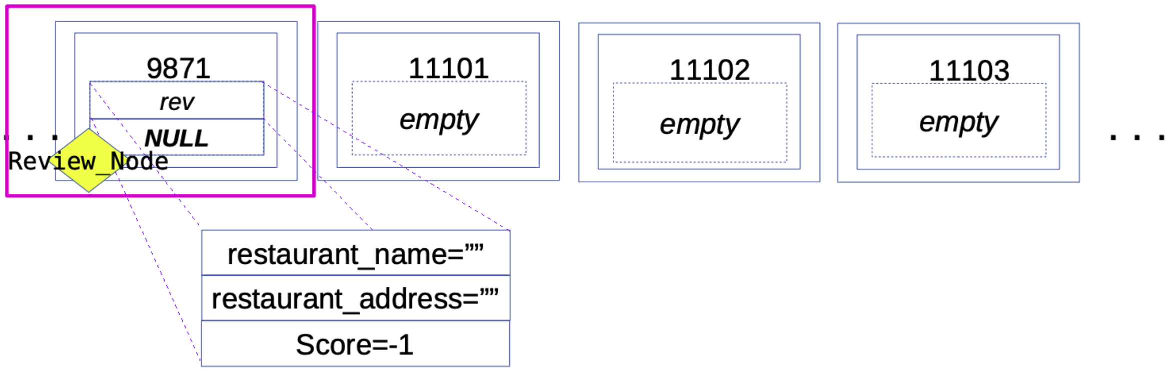


Things to note:

- The newly reserved *Review_Node* is shown inside the purple box.
- It *doesn't have a name tag* because it is not a *variable we declared in the program.*
- Because it doesn't have a name tag, the only way to get to it is by having its address (#9871) in a pointer. That's why the call to *calloc()*
  `new_review=(Review_Node *)calloc(1, sizeof(Review_Node));`
  returns the address of the newly reserved space. Our pointer *'new_review'* has the address of the newly created node.
- The new node *does not belong to any function.* It is *not a local variable.*
- The new *'Review_Node'* has two fields, as expected: one field of type *Review* that we called *'rev'*, and a pointer to the next node in a linked list, we called that pointer *'next'*.

The next few lines in *new_Review_Node()* initialize the contents of the new node to show that it's not been filled with valid data

```
new_review->rev.score=-1;
strcpy(new_review->rev.restaurant_name,"");
strcpy(new_review->rev.restaurant_address,"");
new_review->next=NULL;
```

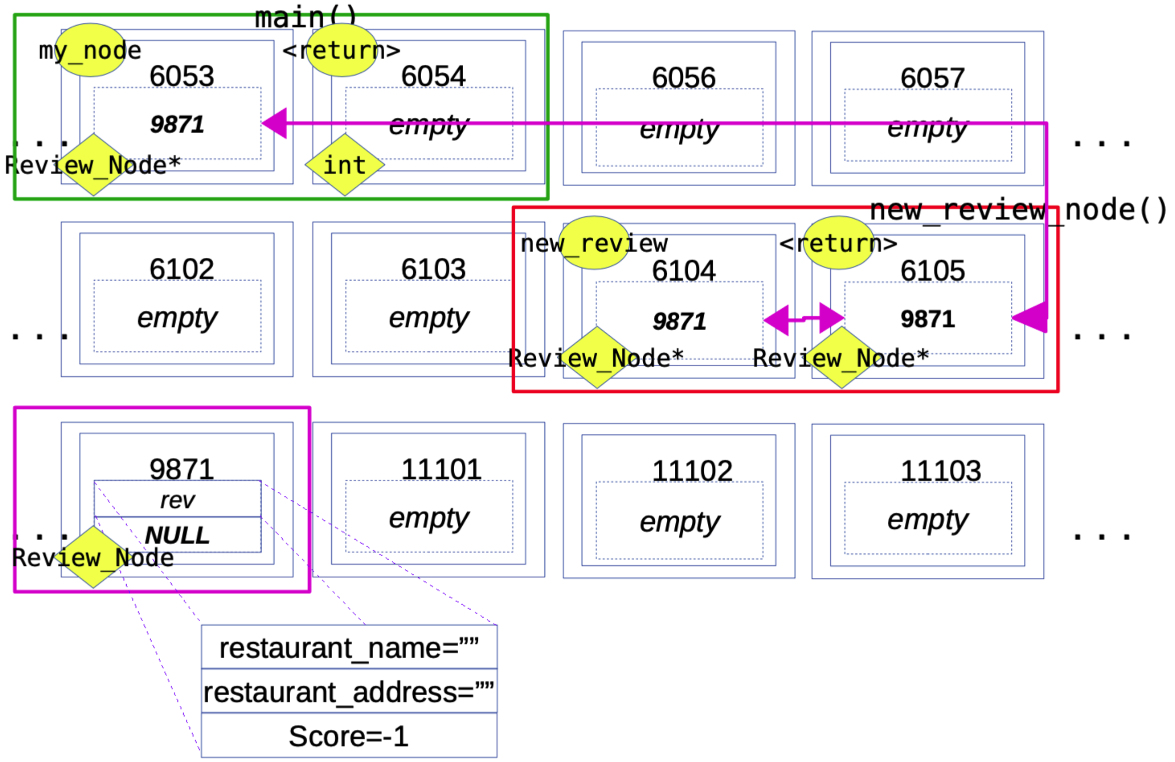


As you can see in the diagram above, *'rev'* has its *three fields* and all of these are initialized to reasonable values that show the node has not been updated with real data.

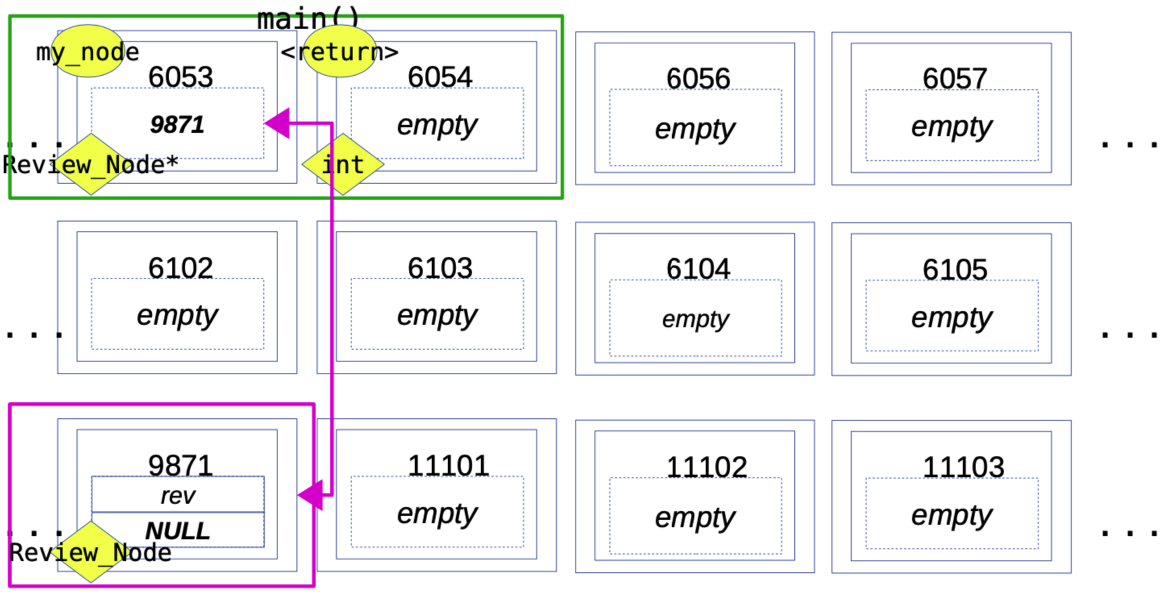
Finally, *new_Review_Node() returns a pointer to the newly created node.* This means copying the address of the node to the *return value*, and then assigning that to *'my_review'* from *main()*

```
my_node=new_Review_Node();
```

So the contents of memory *at the moment the call to new_Review_Node() returns the pointer to the new node* look like:



So now *main()* has a pointer to the newly allocated *'Review_Node'*. Finally, the space reserved by *new_Review_Node()* is released for re-use (you know this happens after a function ends).



Things to note:

- Once the call to *new_Review_Node()* is completed, we have a new *'Review_Node'* somewhere in memory, initialized to show it's currently empty of actual data.
- *main()* has a pointer to this node so it can access and change information within
- Even though the new node was created by *new_Review_Node()*, it doesn't get removed when that function ends. The new node does not belong to any function, and it's not a local variable. *It will stick around until we decide to release the space it uses.*
- Because the new node doesn't have a name tag, the only way to get to it is with a pointer.
- As a result, *if we lose the pointer, we can never find this node again.*

The next few lines in *main()* use the pointer we have to assign meaningful values to the *fields of the restaurant review stored in the node.* Let's look at just one of them:

```
my_node->rev.score=3;
```

The notation here deserves a moment of attention. First, *'my_node'* is a pointer to a *'Review_Node'*. The review node has *two fields: 'rev'* which is the actual review, and *'next'* which is the pointer to the next node in a list. We want to update the contents of the review in *'rev'*, and we are using a pointer, so to access *'rev'* we use

```
my_node->rev
```

but that is not enough because *'rev'* itself has *three fields: restaurant name, restaurant address,* and *score*. Because *'rev'* is a variable (not a pointer!) we use the dot '.' operator to access its fields. So, putting everything together, the line

```
my_node->rev.score=3;
```

can be read as *"go to the 'rev' field of the node whose address stored in my_node, update the 'score' field within 'rev' to be equal to 3".*

Have a look at how *main()* updates the restaurant name and address, and how it prints the contents of the review. All using the pointer we have to this node.

Once *main()* is done working with the node, there's one little detail left to take care of. Just before *main()* exits, we need to return the memory that was reserved for our *'Review_Node'*.

```
free(my_node);
```

This tells the computer we are done using the space reserved at the address stored in *'my_node'* and want to release this space. ***You should always make sure you release (free) all the memory you have requested with calloc().*** Not releasing memory you acquired is called a ***'memory leak'*** and can get you in trouble by using up your computer's memory.

Summary up to this point

- We have created a compound data type called *'Review'* to store restaurant reviews.
- We have created a *'Review_Node'* data type which can be used to build a linked list of restaurant reviews
- We have implemented a function to reserve space for new *'Review_Nodes'* on-demand, and we have seen *in detail* how this function works, how space is reserved, and how we use pointers to access memory that has been reserved *on-demand*.

ASIDE

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 you really understand what is going on when we reserve memory on-demand, or the process you have to follow if you ever have to (and you 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, and making sure you see that you can understand every single step, that the steps all make sense, follow logically from what needs to be accomplished, and that you can visualize what is happening in memory as you are going through the process of allocating and initializing a new data item on-the-fly.

At this point, given all the examples we have done of how variables, pointers, compound types, and function calls are processed, you should have a pretty good understanding of what happens when you 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 looking at more complex programs and looking at every single step in them would take more than the entire length of your degree!

ASIDE (CONTINUED)

But do not forget – C is a very simple and straightforward language that doesn't do anything you didn't ask it to do. You can ***always*** figure out exactly what is going on if you think in terms of boxes in memory that correspond to the data your program is working with, and operations on these boxes. Whenever you're not sure what's happening, *take a blank sheet and a pencil and draw a memory diagram, and make sure you understand what your code is doing!*

EXERCISE

Write a little program that

- Creates an *int_node* data type, 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* while 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 exiting.

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 compound data type to store the data for a single review
- We know how to define a linked list node type 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:

- Code to initialize a new (empty) linked list
- Code to insert a newly created review node in the linked list
- Code to print the reviews in the list whenever we want

- Code in *main()* to allow the user to input as many reviews as desired

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

Write in *pseudocode* the steps we need to implement in *main()* so that our program

- Gives the user a choice of
  a) Entering a new review
  b) Printing out all the reviews entered thus far
  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.

Let's have a look at how we would implement the steps above 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. It contains the different sections we need to complete to implement all the functionality requested. *However*, it does two important things:

- It declares a new, empty linked list. The line

Review_Node *head=NULL;

declares a *pointer* to a *'Review_Node'*, and it sets that pointer to ***NULL***. This is how we create an empty linked list in C!

Question: How do we check if a linked list is empty?

- It provides a loop that prompts the user to choose a number from 1 to 3. Depending on the user's choice, different code is 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?

So the core of our program is already there.

ASIDE

When you are writing a complicated program, it is a good idea to write a *little driver program* that has a loop like the one above, and allows you to test different components of the program separately, and to choose which one gets tested, in what order to test the components, and what information to pass to each of them.

Let's now fill-in the details. First, let's have a look at the code for option '1', it should insert a new review into our linked list.

```
if (choice==1)
{
    // Get a new review node
    one_review=new_Review_Node();

    // 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'*.

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 contained in the *'Review_Node'*. *Remember!* We have bento boxes inside bento boxes: The *'Review_Node'* contains a *'Review'* called *'rev'*, and that review contains a *name*, and *address*, and a *score* for the restaurant.

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 talked above about 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.* That means 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.

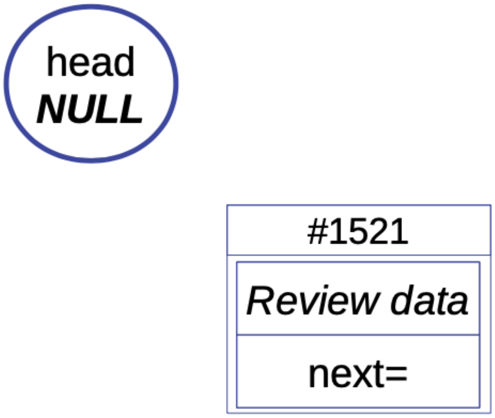
Example: Show in a diagram what the linked list looks like after we insert two reviews. The list is initially empty.

1. Initial state:

   We have a *pointer to the head of the list*, but initially it is ***NULL*** so the list is empty.

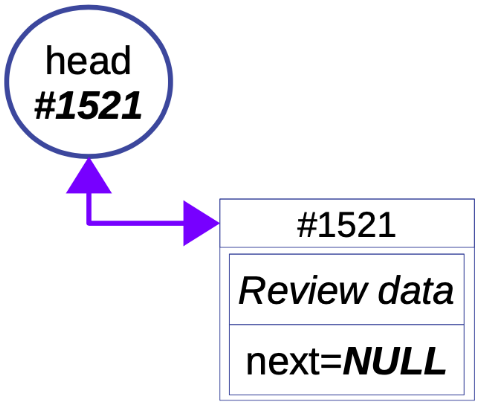
2. Inserting the first review node.

   Our program will allocate and fill-in data for a new review



the new *'Review_Node'* already contains a valid review, input by the user. We need to link it to the list. Remember the process for linking a new node at the head of the list:

- We copy the address currently in the *head pointer* to the *'next'* pointer in the *new node*
- We copy the address of the *new node* (we got a pointer to it!) to the *head pointer*.



We now have a linked list! It has a single node, ***the head node***, but it is a proper linked list.

Things to note:

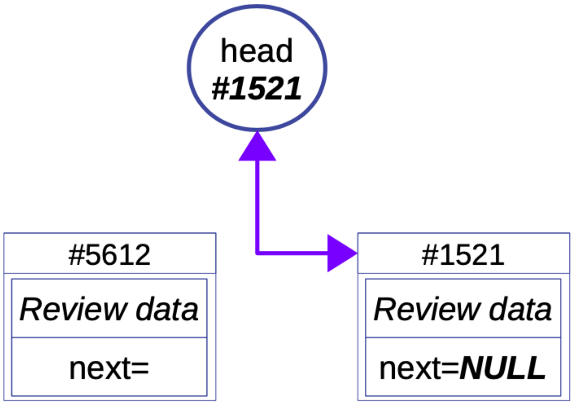
- The *'next'* pointer in our ***head node*** is ***NULL*** because we copied the previous head pointer which was ***NULL*** onto it.
- The ***head pointer is not a node in the list***, it is just the address of the first node in the list.
- ***Do not confuse the head pointer with the head node!***

IMPORTANT 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 having weird behaviour in code that uses linked lists ***check that the list is properly terminated.***

3. Insert a second review into the list.

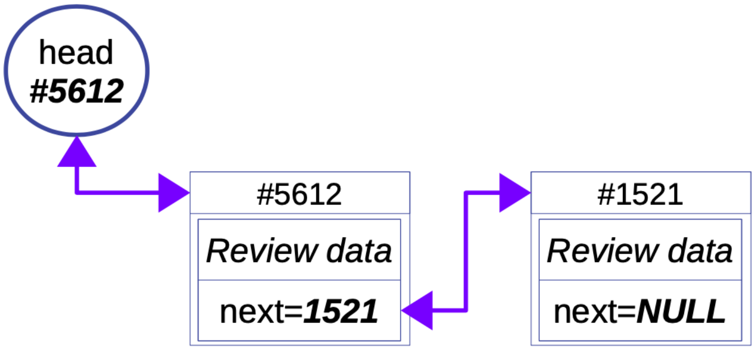
Once again, our program will allocate and fill-in a new *'Review_Node'* with data. Just before we link the new node to the list, we have this situation:



Once again we

- Take the address in the *current head pointer* and copy that to the *new node's 'next'* pointer
- Then we copy the address of the *new node* to the *head pointer*

This leaves our list looking like this:



The same process will allow us to add as many nodes as we want to our linked list.

EXERCISE

Show what the list would look like after we insert two more reviews, the first one has an address #3141, and the second one has an 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;
}
```

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 contained in the pointer *'new_node'*.

EXERCISE

Draw a memory diagram that shows what happens when we call the *insert_at_head()* function. Your diagram should show

- The *head* pointer variable from *main()*
- The current *head node* somewhere in memory
- The *new_node* pointer variable from *main()*
- The *new node* somewhere in memory
- The parameters and return value for *insert_at_head()*
- The *final* values for all pointers (after the call to *insert_at_head()* is completed)

That completes option '1', inserting a node into the list. Let's see now how we could implement option '2' – printing all the reviews currently in the list.

ASIDE

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*. Make sure you fully understand how it works!

Traversing a linked list

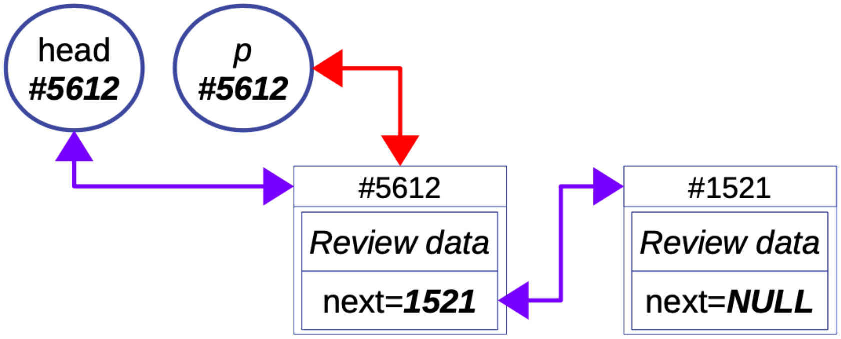
The process requires you to:

- Set up a *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:
  - Processes the node whose address is in the *current traversal pointer*
  - Updates the *traversal pointer* to point to the next node in the list
  - The loop ends when the *traversal pointer* is ***NULL***. This indicates the end of the list has been reached.

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)
    {
        // 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 we were using above:



The print function declares a *traversal pointer* called *'p'*, and initializes it to point to the *head node* (the address of which was provided by the *head* pointer: #5612).

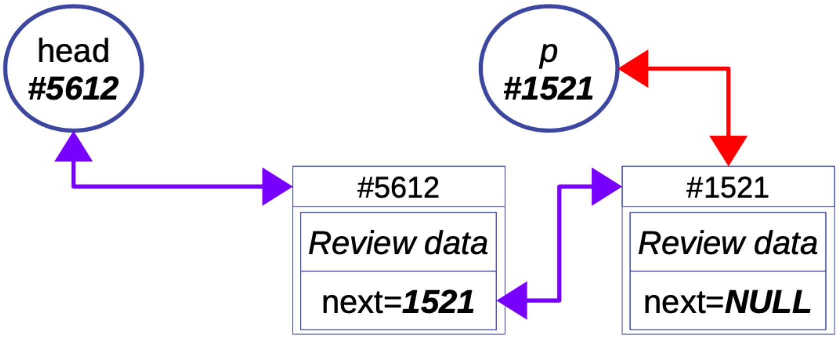
Now the function goes into a loop:

- Print out the contents of the node *'p'* points to (#5612)

  <information for one restaurant is printed>

- Then it updates *'p'* with the address in the *'next'* pointer at the node (#1521)

So we have:



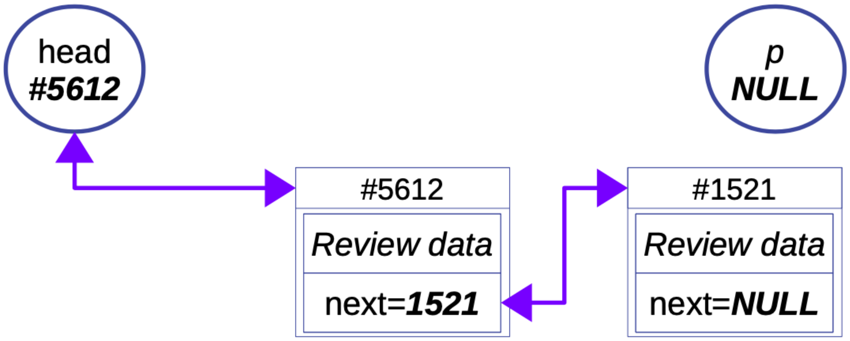
We go through the loop again:

- Print out the contents of the node *'p'* points to (#1521)

  <information for a different restaurant is printed>

- Then it updates *'p'* with the address in the *'next'* pointer at the node ***(NULL)***

And so we have:



At that point the loop exits (we have reached the end of the list).

What you should take from this:

- Traversing a linked list involves using a *traversal pointer* that is updated to point to each consecutive node in the list.
- The traversal process is straightforward:
  - Initialize the *traversal pointer* to the *head node's address*
  - Loop until the traversal *pointer* is ***NULL***
    - Carry out the desired operation on the node
    - Update the *traversal pointer* to point to the *next node*

You will be doing this a lot! So it pays to make sure you have a very solid understanding of the traversal process.

Question: What happens if we pass an *empty list* to the print function? Does it work just fine or will it crash our program?

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 application of the list traversal process* 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;
    }
}
```

You should recognize all the steps of the list traversal process. The only odd detail is that we have two pointers, *'p'*, and *'q'*. Why is this needed?

- The loop will *free* the memory allocated to the node whose address is in *'p'*.
- However, the *address of the next node* is stored in the node we are about to remove!
- If we tried to access *'p->next' after we free* the memory for this node, our program would crash!
- So we use *'q'* to temporarily store the address of the *next node* in the list. We can then remove the node, and update the *traversal pointer* with the address we saved in *'q'*.

Putting everything we have built above into a complete program, we get the listing below.

```
/*
    CSC A48 - Unit 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) 2018 - F. Estrada & M. Ahmadzadeh.
*/

#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;
```

```c

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));

    // 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;
    }
}

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;
    }
}

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)
        {
            // Get a new review node
            one_review=new_Review_Node();

            // 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);
        }
    }

    // User chose #3 { Release memory and exit the program.
    delete_list(head);
    return 0;
}
```

What we have accomplished up to this point

At this point, you know how to build a linked list to contain items of a compound 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! You will find linked lists in a variety of flavours, and in different programming languages. So, remember the following:

- The organization of a linked list is the same regardless of what it contains, what programming language you're using, and what application it's meant to support.
- The same is true for the process of inserting new nodes into the list. The actual implementation will change depending on your programming language, but the steps are the same.
- The process of *list traversal* is also independent of programming language, list contents, or application.

So, make sure you have understood the *concepts and process* behind these three things, they are the most important part of the work we've done up to now.

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 you understand 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.

10 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 this term 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
- Search 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 you 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.

Within 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 course, 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* for 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).

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 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, const 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.

IMPORTANT NOTE

1. 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.

2. You may have noticed the function declaration has a *keyword we haven't seen before:*

```
Review_Node *search_by_name(Review_Node *head,
                            const char name_key[])
```

   the part where we declare the *search key* to be *'const char name_key[]'*. As you know, strings are just *arrays of chars*, so we can pass a string into the search function by declaring a parameter *'char name_key[]'*. However, you also know that if we give a function a pointer to an array, *the function can go and change the contents of that array!*

   Since we *expect the search function to* ***not*** *modify the search key*, it is good programming practice to write the function so that it declares its input parameter to be *'const char name_key[]'*. The ***'const'*** keyword specifies that the contents of that array are *constant and can not be changed within the function* (this idea should be familiar to you from A08 and Python – a data item declared *'const'* in C is *immutable*. However, in C no data type is inherently mutable or immutable, it's up to you to decide when and how to use *'const'*).

   Making the array *constant* achieves two things:

   - It guarantees to anyone using your code that the *search_by_name()* function will ***not*** change any string you pass into it.
   - It makes sure you don't introduce a bug by changing an input argument that is not meant to be modified by your function.
   - The *'const'* keyword can be used with any data type, including compound types declared by you.

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();
```

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);
        getchar();
    }
}
```

Adding these improvements to our code allows us to update reviews for restaurants already added to our linked list. The complete program listing appears below:

```
/*
    CSC A48 - Unit 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) 2018 - F. Estrada & M. Ahmadzadeh.
*/

#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));

    // 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;
    }
}

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;
    }
}

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!
```

```c
}

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!=4)
    {
        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();

        if (choice==1)
        {
            // Get a new review node
            one_review=new_Review_Node();

            // 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();
            }
        }
    }

    // User chose #4 { Release memory and exit the program.
    delete_list(head);
    return 0;
}
```

EXERCISE

1. Compile and run the code above, 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 change a review for a restaurant that does not exist (yet) in the list
    - 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.*

2. 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 allow the user to do this, 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.

3. Write a search function that *prints all restaurants with a review score >= than the specified search key value.* This would be useful if a user wanted to check out restaurants with a score equal to or greater than a specified value.

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. the case for which our program has to do the most work!), 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 you should take from the above:

- Searching on a linked list is just a *list traversal*, checking each node for 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 (remember the steps you had to do to find the t-shirts in your collection of lockers in Zurich!)
- We say that the *search* operation has *linear complexity* on a linked list. That means that the amount of work done by the search function can be described by $k * n$ where k is some constant and n is the number of nodes in the list.

An algorithm or function with *linear complexity* is not too bad as far as things go, but if you think about very large collections of data (e.g. the millions upon millions of documents indexed by

Google), it should be clear to you that a linked list will simply not be a fast enough data structure for organizing data. 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* on 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).

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. But think about this:

- 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)
  - If there are multiple matches to a *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 the *schema* of the database – that is, the list of *fields* that contain the information the database will record and manipulate, and the list of *keys* that can be used to search for information within the database.

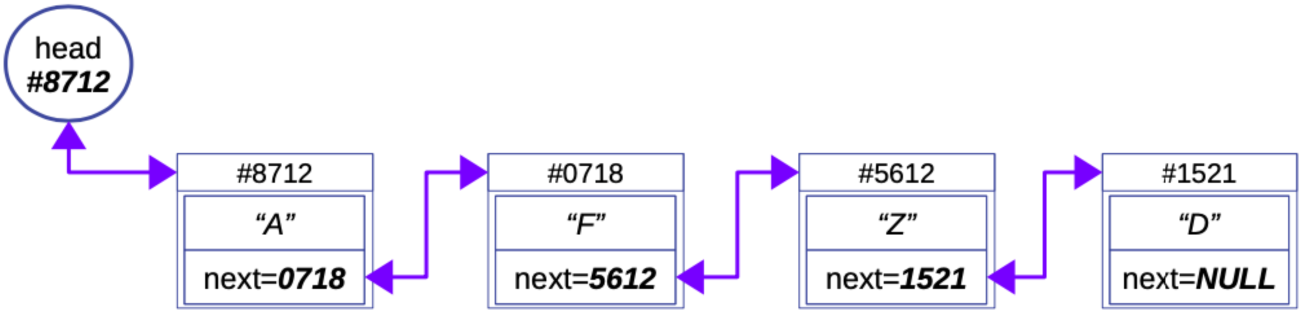
Generating *unique keys* that identify each item in a database is crucial for maintaining information consistency – there is a reason you get a *unique student number* when you join the University!

You can learn a lot more about *keys, records,* and *databases* if you take our database course CSC C43. For the time being, it is enough for you to think about what your program should do if the user wants to insert multiple reviews that have the same restaurant name (at different addresses), or different name but same address.

11 Deleting nodes from a linked list

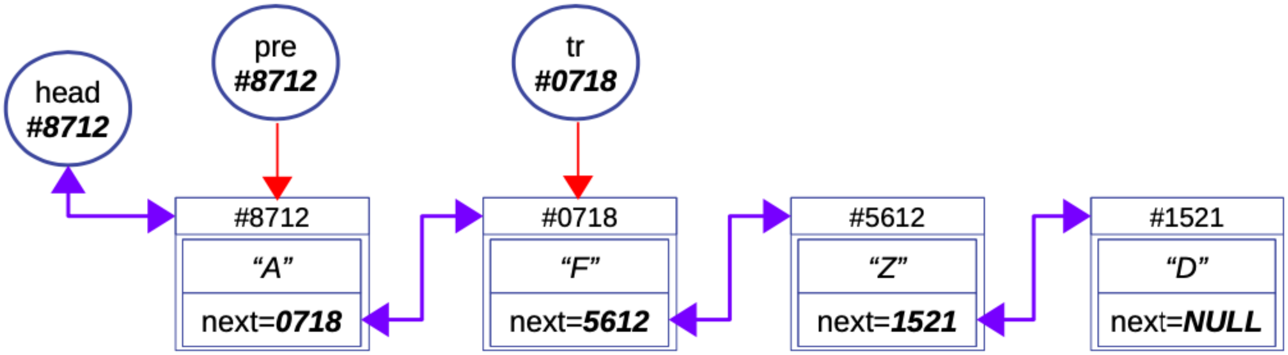
The last operation we need to implement to complete our linked list is the *delete* or *remove* operation. As the name implies, it removes a specific node from a list. Because it looks for a specific item, it involves a *slightly modified search process* – so it is in essence a *list traversal* operation.

Let's first have a look at a diagram that shows what we need to do if we want to remove a particular node in a linked list:



Suppose we want to *delete* the node that contains the item *"Z"*. We will do this using a slightly modified *list traversal* that uses two pointers to find the *node we want to remove*, and its *predecessor*. We need to keep track of a node's predecessor because we need to link it to the node just past the one being removed. In the example above, we are removing the node with *"Z"*, and we need to link its *predecessor* (the node with *"F"*) to its *successor* (the node with *"D"*).

The traversal starts with the traversal pointers as shown below:

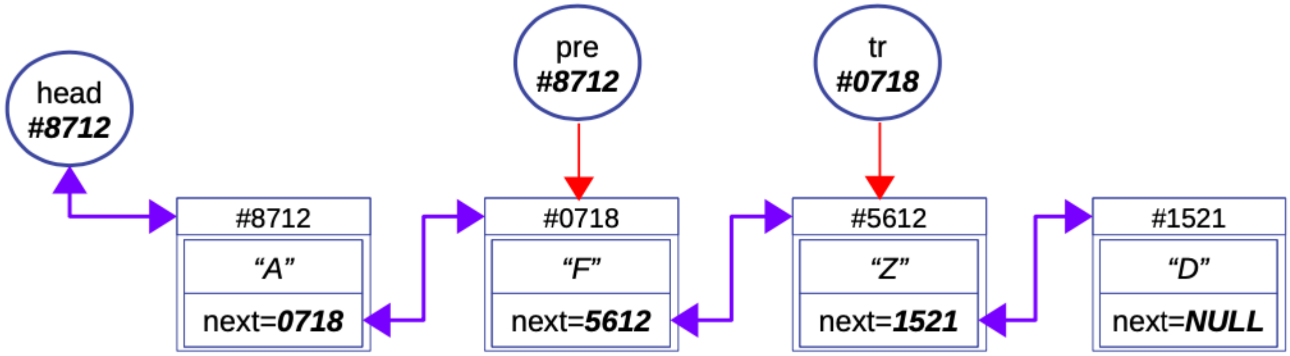


The *predecessor pointer 'pre'* points to the *head node*, the *traversal pointer 'tr'* points one node after *'pre'*.

We then *loop until we find the node we want, or 'tr' is* ***NULL***

- Check if the node at *'tr'* contains the item we want to remove
  - if it does, we remove the node and exit the loop
- Otherwise, move both *'pre'* and *'tr'* to the next node.

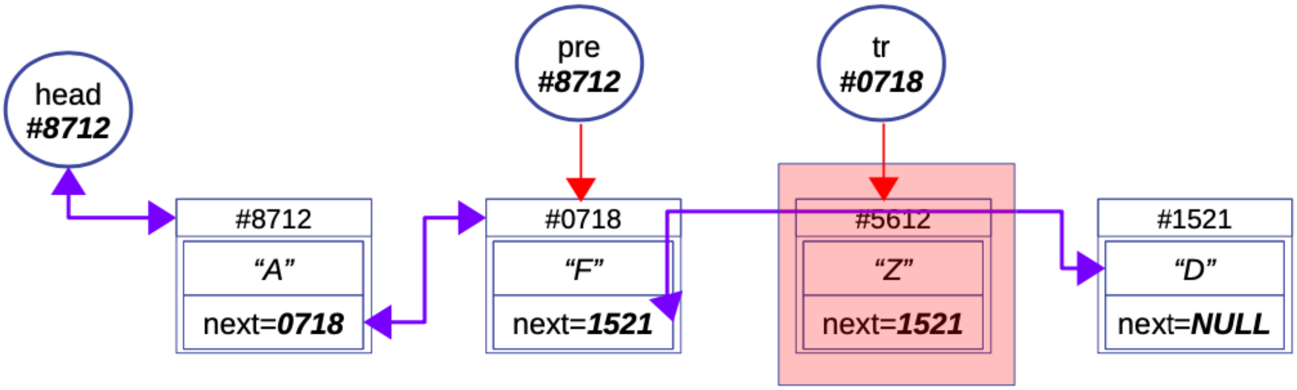
In the case above, we are looking for the node with *"Z"*, it's not at *'tr'* so we move both pointers to the next node:



Now we have the node with *"Z"* at *'tr'* so we proceed to remove the node. This involves:

- Copying the *'next'* pointer from the node at *'tr'* to the *'next'* pointer at *'pre'*. In the example above, it copies #1521 to the *'next'* field at the node with the *"F"*. This effectively links the *predecessor* and the *successor* of the node being deleted.
- We then *free* the node being deleted.
- We exit the loop

This leaves our list in the following state:



The list remains properly linked, and the deleted node is shown in red.

The process works for any nodes *other than the head node.* The head node is a special case because *it doesn't have a predecessor!*

Question: What is the process for removing the *head node* of the list?

Let's see what the function for removing a node looks like:

```
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
}
```

The code above implements the list traversal we looked at, with a *predecessor pointer* and a *traversal pointer*. It 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*.

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 means one more choice from the menu, and code to call the delete function:

```
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);
}
```

The complete listing for the program can be found at the end of the notes for this section. Try it out!

EXERCISE

1. Write a delete function that allows a user to delete reviews by specifying the restaurant address.
2. 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, like 1).

12 A variation on the List ADT

Before we close this section, it's worth exploring an important variation of the *List ADT*. It provides the right form of organization for a wide range of interesting applications.

Queue ADT

A Queue ADT defines a collection in which items are *sequentially ordered* (like in a list). The collection supports the following operations:

- *Enqueue (insert)* – adds a new item *at the end of the queue*
- *Dequeue (remove)* – removes the item currently *at the front of the queue for processing*

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 *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.

Finally, queues are essential for applications that use *graphs* to represent and process information. *Graphs* are data organization/storage structures in which items are represented by nodes, and nodes can be linked to each other in a way that encodes specific relationships between data items. One example are *social networks*, these have nodes for *users*, and the links indicate connections between users.

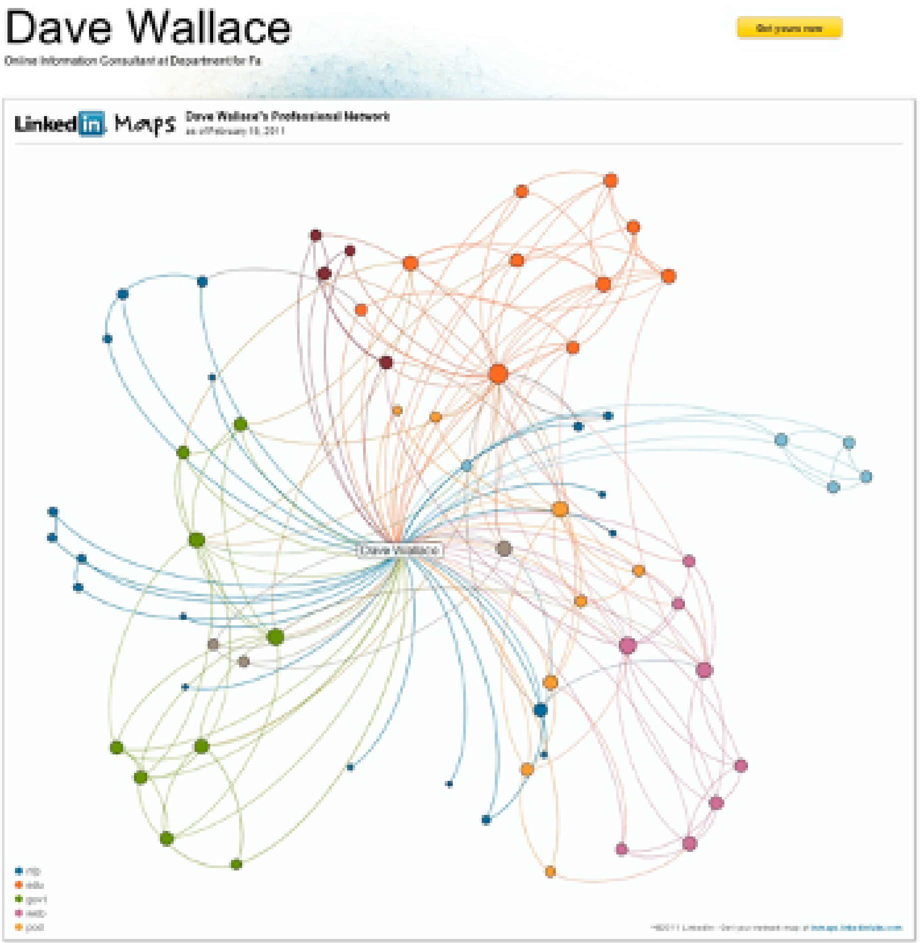


Figure 5: A sample of a professional network, showing connections between one user and others, colour coded by domain. Photo: Dave Wallace, Flickr, CC-SA 2.0

Processing information in graphs often involves placing nodes in a queue. Artificial Intelligence search methods also use queues to accomplish tasks such as 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!

And now for the best part of this section: ***You already know how to implement a Queue ADT!***

EXERCISE

Think about how you would implement a *Queue ADT* using the *linked list data structure* we developed for the *List ADT*.

13 Wrapping up and summary

What you should have learned after studying these notes and completing all the exercises

- You should be able to explain why we need to think carefully about how to store and organize collections of data.
- You should be able to explain why we need to be able to reserve space for data items on-demand (i.e. you should know what the limitations of arrays are and why we can't use them for applications where the amount of data is not specified at the start).
- You should know how to create compound data types (bento boxes!) for data so we can represent complex items.
- You should know how to create and manipulate variables and pointers for complex data types.
- You should be able to explain what a container is, and how we use it to store, organize, and access collections of items.
- You should know what a *List ADT* specifies, how it organizes data, and what operations it supports.
- You should understand how a linked list works, conceptually. What it takes to search for an item, add an item, and delete an item in a linked list.
- You should be able to implement a *linked list data structure* in C. This includes:
  - Defining compound data types to hold the information we need
  - Define 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
- You should understand how much work is involved in list traversal. You should be able to explain why we say that the amount of work for *traversal* is *linear* w.r.t. the number of nodes in the list.
- You should be able to explain the difference between an *ADT* and a *data structure*.
- You should be able to go through the program listing at the end of this section and understand everything it's doing.
- In addition to that you should have learned the aspects of C implementation that we used to implement 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 you to test parts of your code
- How and when to pass and return pointers so functions can work on linked lists

Make sure you have achieved all of the above learning goals for this section. If anything is not clear, please visit your TA or course instructor during office hours and bring all your questions!

Why this section 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 collections, which is a general principle that applies to information storage for pretty much every application that uses computers to manage information. 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.*

ASIDE

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 course.

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 you 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 data.*

This will be the topic of the next unit of the course. For now, let's see what kinds of problems we can solve having learned about containers and lists!

Problem Solving

As we said at the start of the course, A48 is about learning general techniques used for solving problems in computer science. In this unit, 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.*

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, make sure to ask your TA or course instructor.
- 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 your assumptions 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 *accessed* by whichever code needs 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 thought 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 an already existing solution.

What to do when you run intro trouble:

- Figure out *what part of the material* you're having trouble with.
- *Review* that material in the notes, carefully. Check you understand things by writing down an explanation for someone other than yourself.
- When you get stuck with something, come and talk to us in office hours! We're there to help!

Note: We will ***not*** provide solutions to all the problems below. They are supposed to be *for you to work out.* However, some of the problems will be *discussed in tutorial*, and we will ***provide all the help you need*** while you're working on your solution. The weekly *practical sessions in the lab* are the perfect space for you to work on them, with access to a TA that can help clarify any parts of the problem you're having trouble with, or can assist you with technical aspects of implementing your solution. *Make the most of the practical sessions! But do not ask your TA for the solution, and do not post solutions to the forum.*

Remember: these problems are intended to make you think, and figure out what material you still haven't mastered! – ***DO NOT STRESS*** if they seem challenging. They are, but with a bit of guidance and focused studying you will be able to think of a way to approach, and eventually solve them.

Problems involving containers and lists

P0 – 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).

a) *Computing the length of a linked list*

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). Assume nodes in the list have a pointer 'next' to the next entry in the list, just like all the examples we did above.

P1 – 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'* in the linked list contains:

```
Item      item_info;
Item_Node  *next;
```

The 'Item' itself 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
```

a) *Complete the definition of the 'Item' and 'Item_Node' in C*

This is basically to practice your grasp of the syntax needed for defining new data types.

b) *Implement the 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.

P2 – You have found a summer job at the central *Toronto Public Library*. The library 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.

Each *'Item_Node'* in the library's list contains:

```
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

    // Here 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 has already cataloged the content at each branch, and stored it in a (you guessed it!) *linked list.*

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.

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. Instead, they put everything into the *'Item_Node'*. Write down what you believe would be:

Advantages of representing items in this way:

Disadvantages of representing items in this way:

P3 – You're working on an <u>*open source*</u> 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.

a) *Implementing a function that builds a sorted linked list*

Write down the steps required to

- Take 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, or ask a TA to do a demo with playing cards!)

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).

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.

P4 – You have been hired for a Co-Op placement at the University Health Network. Having seen that you learned C during your A48 course, they decide 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.

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 unit's material
- You will practice every concept covered in this unit, and apply it to problem solving
- You will practice implementing in C code that deals with compound data types, and data collections.

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 (what data structure to use to keep the information), and 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

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.

1) 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.
2) Indicate *special constraints* you can identify for each field (e.g. range of values, uniqueness constraints, etc).
3) 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
4) Write an implementation in C of your data representation model.

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.

You need to provide:

- An overall description of the solution:
  - What data structure(s) you will use, explaining why you need those
  - 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
  - A pseudocode description of the part of your solution that adds a new patient
  - 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! What remains is implementing the solution.

c) *Implement and test your solution!*

You can bring your implementation (or partial implementation) to a practical session to show to (or get help from) your TA.