Edinburgh Napier University

SET08701 C Programming

Workbook 2

GNU Toolchain Version

Dr Kevin Chalmers

Contents

List of Figures

List of Algorithms

Unit 1

Including Files and Declaration Order

We have now covered how data is represented on the machine when working with C, and we have also examined how basic C code is converted to Assembly (and therefore machine) code. Let us now look a bit more into how code is generated and how we break our code up into separate files. We are going to move onto developing larger and larger applications from now on as we will be able split our code up accordingly. First, we need to look into what is known as the *pre-processor*.

1.1 The Pre-Processor

The pre-processor is a part of the code generation step that occurs when working with C based languages. The pre-processor runs before the main compilation as it changes the file that needs to be compiled. It can do this in a number of ways, some of which are compiler dependant. The pre-processor commands we will look at are fairly standard, so will work on any compiler (more or less).

We can now re-imagine our compilation step as follows:

$$\text{pre-processor} \Rightarrow \text{compiler} \Rightarrow \text{linker}$$

This means we can consider the code generation going through the following stages:

$$\text{Original C} \Rightarrow \text{Pre-processed C} \Rightarrow \text{Assembly} \Rightarrow \text{Object Code} \Rightarrow \text{Executable}$$

Pre-processor lines are denoted with the hash sign (#). You should notice that we have already been using pre-processor commands - our `#include` statements are such commands. We will look at what these do shortly. First, we will look at what happens when we define values and perform conditional compilation.

1.1.1 Some Pre-Processor Commands

Before looking at our first application using pre-processor commands let us look into some pre-processor commands. These statements allow us to control some of the compilation of our program in particular ways. It is in fact very common to see pre-processor commands in C and C++ code.

The first pre-processor command we will look at is `#define`. This command allows us to define values which we can then use in our code. The pre-processor will replace any use of the defined name with the given value. Listing 1.1 provides some examples.

```
1 #define TEST
2 #define NUMBER 1234
3 #define NAME "Kevin"
```

Listing 1.1: Using `#define`

On line 1 we define `TEST`. There is no value associated with this definition. If we were to use it in code it would be replaced with nothing. Line 2 defines `NUMBER` and assigns it the value 1234. Any time the pre-processor encounters `NUMBER` it will replace it with 1234. Finally, line 3 defines `NAME` and assigns it the value `"Kevin"`.

Let us look at how this affects the code we write. Listing 1.2 is an example application using `#define` before the pre-processor is run across it.

```
1 // Pre-processor will replace NAME with "Kevin"
2 char *student_name = NAME;
3 // Pre-processor will replace NUMBER with 1234
4 unsigned int student_matric = NUMBER;
```

Listing 1.2: Code Before the Pre-Processor

During compilation, the pre-processor is the first stage to run. It looks at the `#define` statements and uses them to modify the code to be compiled. This generates the actual code that the compiler compiles. This code is shown in Listing 1.3.

```
1 // Actual line compiled
2 char *student_name = "Kevin";
3 // Actual line compiled
4 unsigned int student_matric = 1234;
```

Listing 1.3: Actual Code Compiled

It is just a straight swap. There is no checking of code to see if it is correct by the pre-processor. It simply modifies any place it finds a defined value and replaces it accordingly. The compiler is where the check is made to ensure that the code is correct.

Another use of defined values is in performing conditional checks and compiling different code accordingly. This is a very powerful feature of the pre-processor, allowing you to write code that, for example, can target different platforms. For example, see Listing 1.4.

```
1 #ifdef TEST
2 printf("Test defined\n");
3 #else
4 printf("Test not defined\n");
5 #endif
```

Listing 1.4: Using `#define` for Conditional Compilation

In this example different code is produced by the pre-processor based on whether or not `TEST` is defined. This means different code is compiled based on the defined values. Table 1.1 illustrates the different code compiled based on whether or not `TEST` is defined.

There are a number of different pre-processor commands. We will use a couple of these in the module. Table 1.2 describes the most common pre-processor statements.

Defined Value	Code Compiled
`TEST` defined	`printf("Test defined\n");`
`TEST` not defined	`printf("Test not defined\n");`

Table 1.1: Conditional Compilation

Pre-processor Command	Description
`#include`	Includes (adds) a code from a header file to the code file as part of the code to be compiled
`#define`	Defines a value which is then replaced in the code file when found, or used for conditional compilation
`#undef`	Undefines a value. Can overwrite a `#define` used previously.
`#if`	Used to check a value of a defined pre-processor value.
`#ifdef`	Used to check if a value has been defined
`#ifndef`	Used to check if a value has not been defined
`#else`	Used with `#if` and `#ifdef`
`#elif`	An *else-if* statement
`#endif`	Ends a pre-processor if block
`#pragma`	Tells the compiler that the rest of the line contains instructions. These are generally compiler specific, but we will look at one that is fairly cross compiler.

Table 1.2: Some Pre-Processor Commands

1.1.2 Defining Values at Compile Time

Although we can define a value in our code, it is often better to do this during compilation. This allows us to compile different versions of applications just by changing our compile command. This is done by using the `-D` flag when compiling as shown below.

```
gcc -D<value> <filename.c>
```

Listing 1.5: Defining Values at Compilation

For example, if we wanted to compile a file called `hello.c` and define the value `DEBUG` we would use the following:

```
gcc -DDEBUG hello.c
```

We are effectively (but not really) adding a line to our code which is `#define DEBUG`. If the pre-processor encounters and instance of the term `DEBUG` it will act accordingly.

1.1.3 Using `#ifdef` for Conditional Compilation

OK, let us now put what we have learnt to the test. Listing 1.6 is our example code that you should enter using your text editor.

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv)
```

```
4 {
5 #ifdef DEBUG
6   printf("In debug mode\n");
7 #elif RELEASE
8   printf("In release mode\n");
9 #else
10  printf("What mode am I in?\n");
11 #endif
12  return 0;
13 }
```

Listing 1.6: Using `#ifdef`

It is your task to compile the different possible versions of this code and execute them using the `gcc` command with the `-D` compiler flag. This is the exercise below.

1.1.4 Exercise

Compile the application by defining the `DEBUG`, `RELEASE`, and no values. Compile each into assembly code (32bit is easier) and study the difference between the generated assembly code to assure yourself that the code is not being generated unless required.

1.2 Creating a Header File

The real reason we have been looking at the pre-processor is so that we can start understanding what a header file is. You've been using header files since our very first application (for example `stdio.h`), but our description has been a little vague. A header file is essentially a collection of previously written code (normally just declarations - more on this later) that we want to include in our own code. It can contain any standard C statement or declaration - it essentially allows us to separate our code into different.

Let us start by declaring a new header file. We will call this file `hello.h`. Create this file now.

```
1 #pragma once
2 // This file needs to know what printf is
3 #include <stdio.h>
4
5 void hello_world()
6 {
7   printf("Hello world!\n");
8 }
```

Listing 1.7: `hello.h` Header File

What is `#pragma once`?

We already mention in Table 1.2 the `#pragma` command is an instruction to the compiler and / or linker. The `#pragma once` statement is used to tell the compiler to only include the header file once. This is actually quite important. If a header is included more than once, then the functions and other declarations in the header are also added twice. This leads to a compilation error.

`#pragma once` is a technique to ensure a header is only included once. Another technique is to use *header guards*. A header guard used conditional compilation to ensure that the header is only included once. An example is shown below:

```
#ifndef HELLO_HEADER_GUARD
#define HELLO_HEADER_GUARD

// Code defined here...

#endif
```

When the header is first included, the `#define` is encountered, meaning the `#ifndef` can only be true once. This technique does require more code (and thinking of different header guard defines for each header file), but is technically more platform independent and portable. This is because `#pragma once` is not an official part of the C standard. However, pretty much every compiler supports it.

Now that we have created our header file we can create our main application file. Enter the code below in a new C (a `.c`) file.

```
// Include the hello header.  Note the use of quotes this time
#include "hello.h"

int main(int argc, char **argv)
{
  hello_world();
}
```

Listing 1.8: Main Hello World Application

As you can see, our main application just calls the `hello_world` function defined in our header file. Essentially our application is the same as our original *Hello World* application.

To compile the application just use `gcc` as normal. The header file is automatically included (as it was when we included other header files). The pre-processor generates a single code file for the compiler which takes the following form:

```
#include <stdio.h>

void hello_world()
{
  printf("Hello world!\n");
}

int main(int argc, char **argv)
{
  hello_world();
}
```

Listing 1.9: Actual File Compiled After Pre-Processing

What about `stdio.h`

The code listing above isn't truly what the pre-processor generates. The `stdio.h` file is also added at the top. However in the GNU C library the produced

file is almost 350 lines of code long. Hence we haven't included it here. If you *really* want to see the produced C file from the preprocessor, then run the following:

```
gcc -E -P -o preprocessed.c <inputfile.c>
```

Replacing `inputfile.c` accordingly. You can also add the comments with the `-C` flag.

Declaration Order

Declaration order is an important concept in most languages, but especially in C and C++. In languages such as Java and C#, methods and functions and be declared in separate files and the compiler will work out it all out for you. In C and C++ you have to ensure that something is declared before you use it. This means that sometimes you have to specify that a function or `struct` exists before you explicitly define what it is. In this module we won't encounter this requirement specifically, but you should be aware of this requirement if you carry on through C and C++ programming.

If you run this program you will get the same output at the standard *Hello World* application.

1.3 Compiling Multiple Files into One

Now that we can split our code between multiple files using headers, let us look at how we can split across multiple code files and compile them together to make one application. This is how standard software development works. We break our code up in sensible and logical chunks so that we can reuse, control, and understand our code base. In general we use an *IDE (Interactive Development Environment)* to control this for us, but using make files (hence why we use them) provides the same capability. In fact, and IDE just creates make files for us.

1.3.1 Compiling Multiple Files

Compiling multiple files using the GNU compiler is just a case of providing the code files as a list after the `gcc` command. The following command line gives you the general idea.

```
gcc -std=c99 <filename1.c> <filename2.c> <filename3.c> ....
```

Listing 1.10: Compiling More than One File into an Executable

Defining the `exe` Name

When compiling multiple files together, by default the name of the executable produced is `a.out`. For example, compiling with:

```
gcc hello.c goodbye.c
```

will produce an executable called `a.out`. We can control the name of the executable by using the `-o` flag with the compiler. For example, to control the set the name of the executable above to `myapp` we would use the following:

```
gcc -o myapp hello.c goodbye.c
```

Using a header file, we are affectively creating a bridge between the different code files. The code files contain implementation details which don't need to be known to other code files. The code files just need to know that the functionality exists, which is enabled by the header file. Figure 1.1 gives you the general idea of how headers and header inclusion creates this bridge.

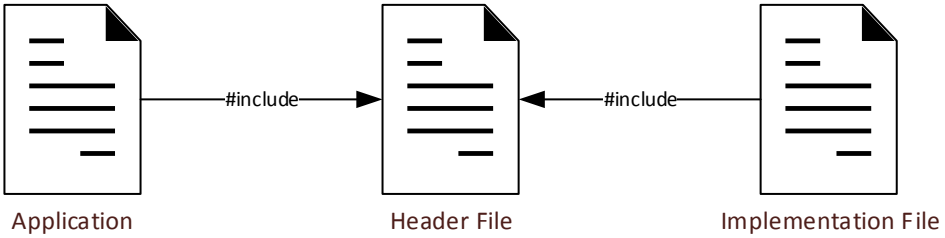

Figure 1.1: Inclusion Structure

1.3.2 Example - Student Details

Let us now build an example that uses separate code files. For this we will revisit our student example. First, let us define our `student.h` header file:

```
1  #pragma once
2
3  // A structure representing a student
4  struct student
5  {
6      unsigned int matric;
7      char *name;
8      char *address;
9  };
10
11 // Declaration of print student method - not implementation
12 void print_student(struct student s);
```

Listing 1.11: `student.h` Header File

Note the use of `#pragma once` again. Otherwise we have two declarations. The first is a `student struct`. This is the same `struct` that we defined before. The second is the `print_student` function. Here we are just declaring the function. We have provided no implementation detail. This is in the `student.c` code file:

```
1  #include "student.h"
2  #include <stdio.h>
3
4  void print_student(struct student s)
5  {
6    printf("Matric: %d\n", s.matric);
7    printf("Name: %s\n", s.name);
8    printf("Address: %s\n", s.address);
```

```
9 }
```

Listing 1.12: `student.c` Code File

Our implementation file just contains the details of how we implement `print_student`. It includes the `student.h` and `stdio.h` header files. Otherwise we are just implementing the same code as before.

Finally our main application file is as follows:

```
1  #include <stdlib.h>
2  #include "student.h"
3
4  int main(int argc, char **argv)
5  {
6    struct student s;
7    s.matric = 123456;
8    s.name = "Kevin Chalmers";
9    s.address = "Edinburgh Napier University";
10   print_student(s);
11
12   return 0;
13 }
```

Listing 1.13: Main Student Application

This is just the same `main` as we developed before. Notice that we have only included the `student.h` header file.

To understand what is happening now when we build the application examine Figure 1.2. At the top of the figure is our `student.h` file acting as a bridge between our `main.c` and `student.c` files. Underneath this is the how the two generated `o` files are linked together to form the main application.

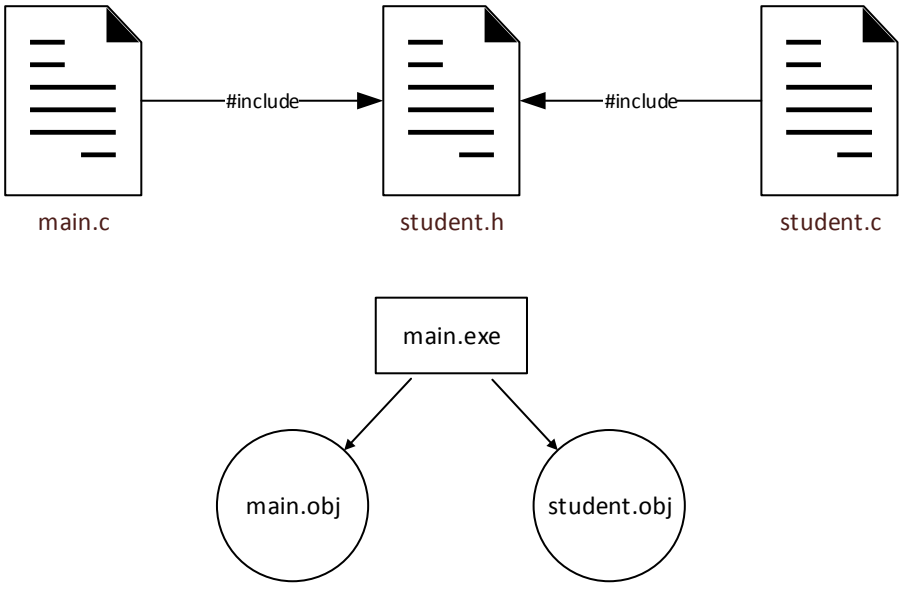

Figure 1.2: Structure of Student Application - both File Inclusion and Object Files

1.3.3 Exercises

1. Compile and build the new version of the student application. You know how to do multiple file compilation, so you should be able to undertake this.

2. Determine what the two code files would look like after the pre-processor pass. This gives you an idea of how the implementation details are separate with the header file acting as a bridge.

3. Make files? Yes we are still going on about these.

1.4 Creating and Linking Libraries

Now we know how to combine separate code files together to form a single application. This is great for code reuse when we have all the code. However, sometimes we might use code from other sources that are already built (so we don't know about the implementation details). This requires us to use other *libraries* in our code.

Library usage is a fundamental part of writing larger applications. You really don't want to compile all the code you need. Sometimes you want to just use pre-compiled code. This is so fundamental to modern software development that you have constantly been doing this since you started programming - you just might not have realised.

Whenever you use built-in functions and code from a particular programming framework (be it Java, C#, or C and C++), you are implicitly using libraries of code. This is just managed automatically for you. What we are going to do is build our own library to give ourselves an idea of the process. Then we will use the library in our code. We will do this in the next section when we build a useful array manipulation library. First, let us look at how we create and use libraries with the GNU compiler tools.

1.4.1 Compiling Code into a Library

The GNU compiler by default will attempt to build an executable from the code you have provided. What we want to do is help us build a library. *Note that a library has no main function.* This is very important to realise. A library is not executable. We simply link it to our applications to build an executable.

The first thing we have to do is ask `gcc` just to compile our code. This is done using the `-c` flag when calling the compiler. This will make the compiler only produce object code. It is this object code we combine together to form a library.

To create a library we use the `ar` command. This command takes a list of object files and produces a `.a` file. The following shows the two steps we need to take to create a library.

```
gcc -c <filename1.c> <filename2.c> <filename3.c> ...
// Compile as many files as necessary into object files
ar rcs <libname.a> <filename1.o> <filename2.o> <filename3.o> ...
// Reusable library of code generated
```

Listing 1.14: Creating a Library from Object Files

With the GNU toolchain, your produced library name must end in `.a` and begin with `lib` (for example `libname.a`, `libmine.a`).

Our header files will still act as a bridge for us as shown in Figure 1.3. This means in C and C++ we require both the header files and the library file to work with a set of pre-built code. The two parts provide the following capabilities:

header files - provide a declaration of the functionality to be provided. It is an interface to the implementation.

library files - provide the definition (or implementation) of the functionality.

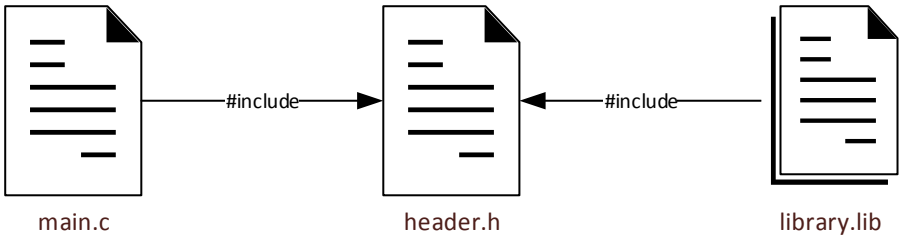

Figure 1.3: Library Inclusion Structure

1.4.2 Linking to a Library

Having a library is one thing, but how do we go about using the library? This is actually managed by the linker (remember the `ld` command - however we will let `gcc` do the work for us). The linker can take more than object code files as part of its set of inputs. It can also take library files. As an example, see below.

```
gcc <filename.o> -l<name> -l<name> ...
```

Listing 1.15: Linking to Libraries

The `name` is the name of the library without the `lib` prefix or the `.a` extension.

This means that our file link would look like that shown in Figure 1.4. It is essentially the same idea as when linking object files in general.

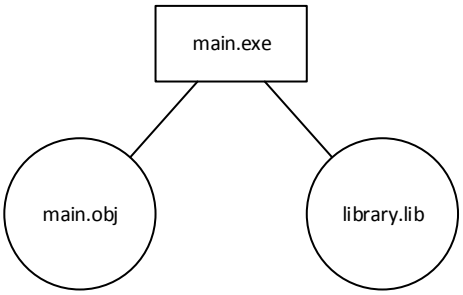

Figure 1.4: Linking a Library

Note with `gcc` and other GNU tools, you have to tell the compiler where to look for libraries if they are none standard. In our case, the location

1.4.3 Exercise - Student Details (Again)

Your task this time is to compile the `student.c` file into a library (which will end up being called `libstudent.`). Once that library is created, create the main application by linking the main `.o` file to the `libstudent.a` file (hint - you will need to tell `gcc` where to look using the `-L` flag. Use -L. to indicate the local directory).

Try and work it out for yourself, but if you are struggling the line of commands is as follows:

```
gcc -c student.c student-main.c
ar rcs libstudent.a student.o
gcc student-main.o -L. -lstudent
```

1.5 A Simple Array Library

OK, now let us develop a nice little reusable library. Our library will allow us to work with an array of data. It will provide the following features:

1. searching an array for a value
2. sorting the array
3. generating random data into the array

We will split these three different functions into three separate code files. Let us look at the these in turn.

1.5.1 `search.h`

Our search functionality is declared in a `search.h` header file. This declares a `search` function that takes the following parameters:

1. the value being searched for
2. the size of the array being searched
3. the array to be searched - remember an array in C is represented by a pointer to the memory location (using the `*` declaration)

The code for the header is below.

```
1 #pragma once
2
3 int search(int value, int size, int *data);
```

Listing 1.16: Search Header File

1.5.2 `search.c`

The `search` function is defined in our `search.c` file. The code will simply look through each value in the array and checking if it is the one required. If it is it returns the index of the found value. If not, it will return -1 (equivalent to not found). Algorithm 1 provides the pseudocode for the linear search algorithm.

Algorithm 1 Linear Search Algorithm

```
1: function SEARCH(value, size, data)
2:     for i ← 0 to size − 1 do
3:         if data_i = value then
4:             return i
5:     return −1
```

The code for the `search.c` file is given below.

```
1 #include "search.h"
2
3 int search(int value, int size, int *data)
4 {
5   // Loop throught data until found
6   for (int i = 0; i < size; ++i)
```

```
7   {
8     if (data[i] == value)
9     {
10      // Found value - return i
11      return i;
12    }
13  }
14  // Not found.  Return -1
15  return -1;
16 }
```

Listing 1.17: Main Search File

1.5.3 `sort.h`

Our sort functionality is contained in the `sort.h` header file, we declares a `sort` function. It takes the following parameters:

1. the size of the array to be sorted
2. the array to be sorted - as a pointer

The code for `sort.h` is given below.

```
1 #pragma once
2
3 void sort(int size, int *data);
```

Listing 1.18: Sort Header File

1.5.4 `sort.c`

Now let us consider how we implement a sorting algorithm. This is quite a fundamental part of computing. There are a number of different searching algorithms, and you will learn about them later in your studies. For the moment, we will do the simplest sort - *bubble sort*. Bubble sort moves values up through the array, *"bubbling"* them up to there position in the array. It does this by iterating through the array multiple times until the array is sorted. Algorithm 2 provides the pseudocode for this capability.

Algorithm 2 Bubble Sort Algorithm

```
1: procedure SORT(size, data)
2:     for i ← 0 to size − 1 do
3:         for j ← 0 to size − (i + 1) do
4:             if data_j < data_{j+1} then
5:                 Swap values
```

Our implementation code for `sort.c` is given below.

```
1 #include "sort.h"
2 #include <stdio.h>
3
4 void sort(int size, int *data)
5 {
6   // Iterate through each value
7   for (int i = 0; i < size; ++i)
```

```
8   {
9     // Loop through values above index i
10    for (int j = 0; j < size - (i + 1); ++j)
11    {
12      // Test if data[j] > data[j + 1]
13      if (data[j] > data[j + 1])
14      {
15        // Swap values
16        int temp = data[j + 1];
17        data[j + 1] = data[j];
18        data[j] = temp;
19      }
20    }
21    // Display % of currently sorted data
22    if (i % 1000 == 0)
23      printf("%.2f%% sorted\n", ((float)i / (float)size) * 100.0f);
24  }
25 }
```

Listing 1.19: Bubble Sort Implementation

Line 22 might seem to be a bit strange. Here we are calculating the % of the array that is sorted (updating this every 1000 iterations). We do this by dividing the number sorted (the `i` value) by the size of the array (the `size` value). This will give us the ratio of sorted values.

> **What is a Bubble Sort?**
> As mentioned, bubble sort is an algorithm to sort data into order. It is one of many such algorithms. It is an algorithm that is *very inefficient*, taking a long time to sort any moderately sized data set. There are far more efficient sorting algorithms that you will come across later in your studies. However, bubble sort is how you would probably sort something small in real life.

Exercise

Work through the bubble sort algorithm, experimenting with how it operates. Write down an array of some values (say 5) out of order and run through the algorithm. Ensure your result comes out ordered, and that you understand what is going on.

1.5.5 `generate.h`

Our final function is defined in the `generate.h` file. This declares the `generate` function. It takes two parameters:

1. the size of the array to generate data into
2. the array to generate the data into

The code for `generate.h` is below.

```
1 #pragma once
2
3 void generate(int size, int *data);
```

1.5.6 generate.c

For our implementation of the value generation code we will simply iterate through each value and assign it a random number. This code is shown below.

```
1  #include "generate.h"
2  #include <stdlib.h>
3  #include <time.h>
4
5  void generate(int size, int *data)
6  {
7    // Seed the random
8    srand(time(NULL));
9    // Generate random numbers
10   for (int i = 0; i < size; ++i)
11     data[i] = rand();
12 }
```

Listing 1.20: Generating Random Numbers in C

Random Number Generation

We have used two new calls in our generate function. The first is `srand`. This stands for *S*eed *Rand*om. A computer cannot create truly random numbers. It uses an algorithm to do this, which requires a starting value. The `srand` function provides this starting value. Any time you call `srand` you change the starting value.

The `rand` function provides a random value. `rand` provides a value within a fixed range. Other random number generators in other frameworks provide different ranges. We won't concern ourselves with this at the moment.

1.5.7 Test Application

Now let us look at our test application for working with the array library. This application will perform the following actions:

1. generate some data
2. print out the first 20 values of the unsorted array
3. sort the array
4. print out the first 20 values of the sorted array

The code for this application is given below. Save this in a file called `test.c`.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include "generate.h"
4  #include "sort.h"
5  #include "search.h"
6
7  #define NUM_INTEGERS 65535
8
9  int main(int argc, char **argv)
10 {
11   // Allocate an array of a given size
```

```
12    int data[NUM_INTEGERS];
13
14    // Generate random numbers
15    generate(NUM_INTEGERS, data);
16
17    // Output first 20 values
18    printf("\nUnsorted\n");
19    for (int i = 0; i < 20; ++i)
20      printf("%d\n", data[i]);
21
22    // Sort the data
23    sort(NUM_INTEGERS, data);
24
25    // Output first 20 values
26    printf("\nSorted\n");
27    for (int i = 0; i < 20; ++i)
28      printf("%d\n", data[i]);
29
30    return 0;
31  }
```

Listing 1.21: Test Application for Array Library

Our inclusion file structure is shown in Figure 1.5.

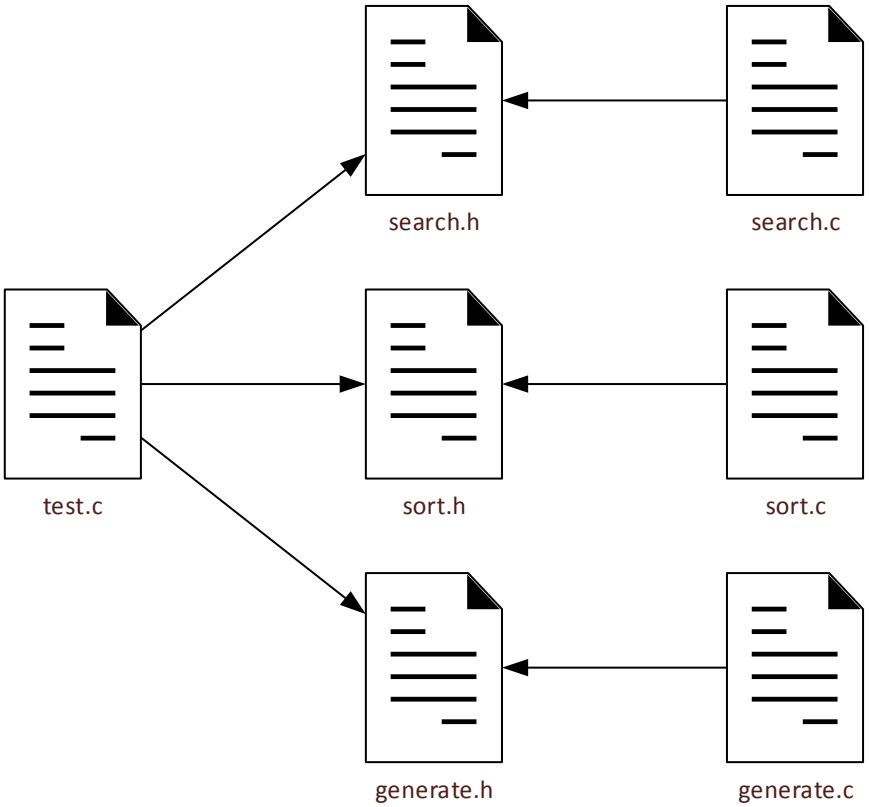

Figure 1.5: Array Library File Structure

1.5.8 Compiling the Array Library

Now let us build our library (hopefully you've entered everything correctly!). We need to go through the following stages:

1. compile the files that make up the library creating the relevant object files

2. use `lib` to create the library file from the object files
3. compile the test application file into object code
4. link the test application object file to the array library

The following is the sequence of steps you need to perform. This would be perfect for a make file!

```
gcc -std=c99 -c search.c sort.c generate.c
ar rcs libarray.a search.o sort.o generate.o
gcc -std=c99 test.c -L. -larray
```

Listing 1.22: Command Line Arguments to Compile the Array Library

1.5.9 Output from Array Library Test Application

An example output from this application is given below. It has been suitably cut down to avoid the many lines of output. Note the % of sorted values and the rate at which it increments the higher the sorted % gets.

```
// ... previous lines

96.13% sorted
97.66% sorted
99.18% sorted

Sorted
1
1
1
2
2
3
3
4
4
5
5
5
5
5
6
7
10
10
10
10
```

Listing 1.23: Output from Array Library Test Application

You now know how to create and use a library. Let us reuse that library as we explore working with file input-output in C.

1.6 Reading Files

This may be a new concept to some of you - how we go about reading and writing files. The principles are actually similar to working with reading and writing from the command line. We are just going to perform the actions with a file.

File I/O is another fundamental part of computing. It forms two thirds of a high level view of an application:

$$\text{input} \Rightarrow \text{process} \Rightarrow \text{output}$$

We will look at both input and output over the next two sections. Let us first look at how we open a file.

1.6.1 Opening a File

To open a file in C we use the `fopen` function. This will return a `FILE*` (pointer to a FILE). The call requires a filename and a mode. The filename has to be a correct filename in the system relative to the place where the executable is run (for our purposes the same folder). `fopen` is illustrated below.

```
1 FILE *file = fopen("filename", "mode");
```

Listing 1.24: Opening a File

The `mode` value is a string telling C how to open the file. There are a few different methods of opening a file. Table 1.3 describes the different methods.

Mode	Description
r	opens the file for reading
w	opens the file for writing. Existing files of the name have their contents discarded
a	opens the file for appending (writing at the end). Will not discard existing file contents. File seeking operations are ignored.
r+	opens a file for reading and updating
w+	opens a file for reading and updating. Discards and existing contents in the file
a+	opens a file for reading and updating at the end. Will not discard contents. File seeking operations are ignored.
b	opens the file as binary rather than text

Table 1.3: File Opening Modes in C

1.6.2 Opening a File for Binary Reading

We are going to read a file in binary. These means we have to combine the read mode (`r`) and the binary mode (`b`). We do this as follows:

```
1 FILE *file = fopen("filename", "rb");
```

Listing 1.25: Opening a File for Binary Reading

What is a Binary File?
A binary file is one where we have data stored in its raw format. In other words, the data is stored in a manner similar to how the computer stores information in memory. This means that numbers are not nice and textual, but rather are stored in their bit pattern form. This can save space, but is not necessarily cross platform.

1.6.3 Reading a Binary File

Let us now write a test application to open a binary file, read it in, and then sort it. To do this we will also write a function that will read in a file and return the amount of data read. The form of the data file will be such that the first 4 bytes will tell us the number of values stored in the data file. This means that we don't

know how many values are stored, so we will have to introduce some strategies for allocating enough space to store our values. The code for our test application is below. We will explain the new parts presently.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include "sort.h"
4
5  // Reads in a block of data as an int array
6  int readfile(int **data)
7  {
8      // Open file for reading
9      FILE *file;
10     file = fopen("numbers.dat", "rb");
11     // First value is number of integers
12     int size;
13     fread(&size, sizeof(int), 1, file);
14     // Allocate memory for that number of values
15     *data = (int*)malloc(sizeof(int) * size);
16     // Read in rest of data
17     fread(*data, sizeof(int), size, file);
18     // Return size
19     return size;
20 }
21
22 int main(int argc, char **argv)
23 {
24     // Declare data
25     int *data;
26     // Read in file
27     int size = readfile(&data);
28     // Sort
29     sort(size, data);
30     // Print first 20 results
31     for (int i = 0; i < 20; ++i)
32         printf("%d\n", data[i]);
33
34     // Free the allocated memory - otherwise a memory leak! (very
           bad)
35     free(data);
36
37     return 0;
38 }
```

Listing 1.26: Reading a Binary File

Where is the `numbers.dat` File?
You will find this on Moodle.

Allocating Memory
We used two new functions in this example - `malloc` and `free`. `malloc` (Memory ALLOCation) creates a block of memory for us to use. This is required as we don't know the number of values we require to store the file contents. We therefore use `malloc` to create the memory block. `malloc` requires just one value - the amount of data (in bytes) we need to allocate. Notice that we used the size of an `int` times the number of values we are going to read.

`malloc` returns a pointer to the memory location it has allocated memory at. This pointer is of type `void` (so we have a `void*`). This means that the type of memory is undefined (it is just a block). We cast it to a pointer to `int` (a `int*`) to set the `data` value.

The other function we have used is `free`. This releases any allocated memory once we have finished with it. *This is very important!*. Let me repeat that - *THIS IS VERY IMPORTANT!*. If you do not free your allocated memory it cannot be used, leading to memory leaks. Over time, this could lead to your application running out of memory. Ensuring you free your allocated memory is an important consideration in C and C++ (there is no garbage collector like in Java and C#). We will spend an entire unit exploring this concept.

Why `**data`? What does that mean?

Look at what the code is saying (remembering that `*` means *pointer-to*). We are effectively saying that we have a *pointer-to* a *pointer-to* `int`. In other words, the `data` value points to a memory location that contains a memory location.

Why do we need this? Well, the call to `malloc` will create a new memory location. If we just used a memory location for `data` (a pointer-to `int`) we would overwrite the memory location in `data` within the function `readfile` but not in the `main` function. We would affectively lose the memory location (and create a memory leak).

At the moment this will seem confusing, but we will spend time exploring this over a couple of units in the module. At the moment, understand that we have passed data as a *pointer-to a pointer-to* `int`.

`fread` and `fclose`

How many boxes do we need after this code? We have two other functions for working with files. The first is `fread`. This reads in data from a file. It takes the following parameters:

1. the location to read the file into
2. the size of the data type being read in
3. the number of values of the data type to read in
4. the file to read in from

Points 2 and 3 above provide us with the amount of data to read in (the size of the type times the number of values).

The second new function we used was `fclose`. This closes the file. *You should always close your files after you have finished with them!*. If you don't close the file you can cause system conflicts. If you are writing to a file, you may lose the information sent to the file when the application exits. The application *will not* automatically push the contents to the hard drive, even on exit. Therefore data can be lost.

1.6.4 Exercise

You should be able to compile and link this file. You will need the `libarray.a` library we generated in the previous section. The application will give a similar output to the last one. However, we are now sorting *a lot* of data, and the application will take time to complete.

1.7 Writing Files

Let us extend the previous version of the application now to also save the sorted data in a text file. The following code will accomplish this for you. You should hopefully understand what is meant by a text file by now.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "sort.h"
4
5  // Reads in a block of data as an int array
6  int readfile(int **data)
7  {
8      // Open file for reading
9      FILE *file;
10     file = fopen("numbers.dat", "rb");
11     // First value is number of integers
12     int size;
13     fread(&size, sizeof(int), 1, file);
14     // Allocate memory for that number of values
15     *data = (int*)malloc(sizeof(int) * size);
16     // Read in rest of data
17     fread(*data, sizeof(int), size, file);
18     // Close file
19     fclose(file);
20     // Return size
21     return size;
22 }
23
24 // Writes strings to the file
25 void writefile(int size, int *data)
26 {
27     // Create file
28     FILE *file;
29     file = fopen("sorted.txt", "w");
30     // Loop through each value, writing to the file
31     for (int i = 0; i < size; ++i)
32         fprintf(file, "%d\n", data[i]);
33     // Close the file
34     fclose(file);
35 }
36
37 int main(int argc, char **argv)
38 {
39     // Read in data
40     int *data;
41     int size = readfile(&data);
42     // Sort
43     sort(size, data);
44     // Write the data
45     writefile(size, data);
```

```
46      // Free the allocated memory - otherwise a memory leak! (very
            bad)
47      free(data);
48
49      return 0;
50  }
```

Listing 1.27: Reading and Sorting a Binary File and Outputting as Text

`fprintf`
We used `fprintf` as a command here. This works exactly as `printf` except that is requires a `FILE*` to print to. Here we have used the opened file, but we could also use `stdout` (the command line).

1.8 Exercises

We've covered a lot of ideas in this unit and you should go through it again to ensure you are comfortable of, and understand the, concepts discussed. File reading we will return to in C++ (where our life gets a little easier). So there are only a couple of exercises here.

1. Write an application that prompts the user for a name and writes it to a file as text. Each name should be on a new line. The application should continue asking for names until END is entered.

2. Write an application that reads in your sorted text file and prints out the values. You should use `fgets` to read in the lines from the text file.

Unit 2

Call Conventions - Passing by Value, Reference, and Pointer

In this unit we are going to look at how we work with functions / operations in more detail. So far, our journey through C has been as follows:

1. Introduction to programming in C, learning how our code is compiled and linked
2. Learning how our data is represented in the computer's memory
3. Learning how our C code is converted into Assembly language (and therefore machine understandable instructions)
4. Learning how our code files are processed and joined together to build compilation units

In this unit we are interested in how our variables are transferred and used by our functions / operations. This involves our first major investigation into what a pointer is and what a reference is. The former concept becomes very important when we look at memory management in the next unit. To use the latter concept, we need to change the language we are using to C++.

2.1 Passing by Value (Copying Data)

We are ready to move onto the main focus of this unit - how values are passed to functions / operations in C (and later if you have time C++). This is where we have to start understanding a little about how values are passed as parameters, and our initial introduction to the stack in the *Inline Assembly* unit will help us here.

Over the next few sections we are going to look at the three techniques for passing a value as a parameter to a function / operation. There are as follows:

1. Passing a value by copying it to the function / operation (*pass-by-value*)
2. Passing a value by providing a pointer to the function / operation (*pass-by-pointer*)
3. Passing a value by providing a reference to the function / operation (*pass-by-reference*)

The first technique we will look at is pass-by-value. This is the technique we have been using in most cases up until now. This technique involves creating a copy of our value(s) and giving them to the function / operation. This means that anything the function / operation does with the value is not reflected in the caller.

With that in place, let us look at our pass-by-value application. This is below.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  void foo(int x)
5  {
6      printf("Start of function, x = %d\n", x);
7      x = 20;
8      printf("End of function, x = %d\n", x);
9  }
10
11 int main(int argc, char **argv)
12 {
13     int x = 10;
14     printf("Before function call, x = %d\n", x);
15     foo(x);
16     printf("After function call, x = %d\n", x);
17
18     return 0;
19 }
```

Listing 2.1: Passing a Parameter by Value

There is nothing unusual or new in this application. The idea we are trying to examine is what is happening in memory and with the variables. We have already covered the stack, and we have mentioned the stack pointers (that tell us the bottom and top of the stack). With that in mind, we can visualise what is happening in our application as shown in Figure 2.1.

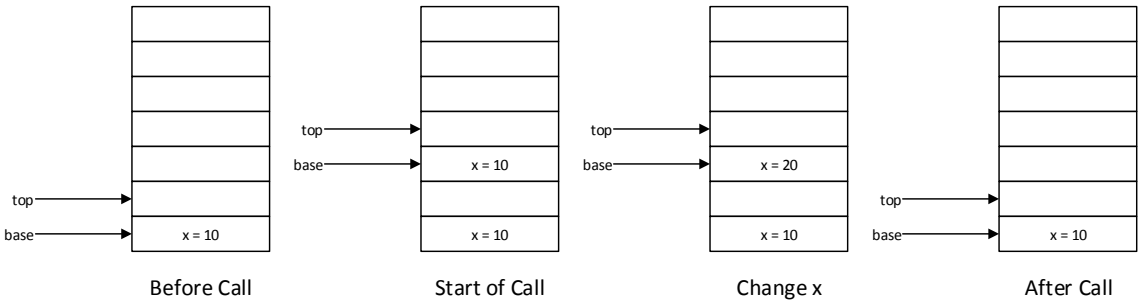

Figure 2.1: Interpretation of the Stack During Pass-by-Value

To start with we have our value that is declared in our main application. This is the value at the bottom of the pointer. When we call the function, we push a copy of `x` onto the stack (let us assume that there is something else on the stack between our two `x` values). When we change the value of `x` in the function, only the function's copy is modified. The original value (at the bottom of the stack) is unaffected. Therefore, when the function exits, this change has been lost.

Running this application will give an output as shown below:

```
Before function call, x = 10
Start of function, x = 10
End of function, x = 20
After function call, x = 10
```

Listing 2.2: Output from Pass-by-Value Application

Scope (yet again)
Basically we are looking at scope here again. The scope of the main application is different than the scope of the function. As such, any variables that are *not references or pointers* will be copies, and be different in each scope. Our work in pass-by-reference and pass-by-pointer allows us to overcome scoping limitations such as these.

2.2 Pointers

Pointers are considered the most difficult part of working with C and C++ as they take novice programmers a bit of time to understand. This isn't helped by the use of similar symbols between references and pointers in C++.

Remember that a pointer is just a location in memory that we treat like a particular value. This is similar to references (see later in the unit), except that we can make a pointer point to different parts of memory as required.

To declare a pointer we use the `*` specifier with the type. For example, a pointer to an `int` is declared as `int*`. To get the address of a variable we use the `&` operator (address of operator). To get the value stored in the memory location pointed to by a pointer we use the `*` operator. Let us summarise this a little. This is given in Table 2.1.

Operator / Specifier	Example	Description
Pointer type specifier	`int *x;`	Declares that a variable or parameter is a pointer type.
Address-of operator	`int *x = &y;`	Gets the memory address of a variable
Dereference operator	`int z = *x;`	Gets the value stored in the memory location represented by the pointer

Table 2.1: Pointer Operators and Specifiers

Let us now implement our *pass-by-pointer* application as we did for pass-by-value. This time we will also print out the memory location of our variable to illustrate that it is not changing throughout the application. The code for our application is below. Note the use of the *address-of* and *dereference* operators.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  void foo(int *x)
5  {
6      printf("Address of x in function = %p\n", (void*)x);
7      printf("Start of function, x = %d\n", *x);
8      // Have to dereference pointer to change value
9      *x = 20;
10     printf("End of function, x = %d\n", *x);
11     printf("Address of x at end of function = %p\n", (void*)x);
12 }
13
14 int main(int argc, char **argv)
15 {
16     int x = 10;
17     printf("Starting address of x = %p\n", (void*)&x);
```

```
18      printf("Before function call, x = %d\n", x);
19      // Have to pass the pointer (or address of) x to the function
20      foo(&x);
21      printf("After function call, x = %d\n", x);
22      printf("End address of x = %p\n", (void*)&x);
23
24      return 0;
25  }
```

Listing 2.3: Passing a Value as a Pointer

Printing Memory Addresses in C

You might notice that we have used a new placeholder - `%p` - to print our pointer. This placeholder will print a memory address. Notice as well that we have to cast the pointer to `void*`. `void*` is the base pointer of all types in C, and therefore we are just saying *treat this as just a raw pointer to memory - no typing.*

An example output from this application is below. Note that the memory address remains the same throughout the application.

```
Starting address of x = 0x7fffab184a1c
Before function call, x = 10
Address of x in function = 0x7fffab184a1c
Start of function, x = 10
End of function, x = 20
Address of x at end of function = 0x7fffab184a1c
After function call, x = 20
End address of x = 0x7fffab184a1c
```

Listing 2.4: Output from Pass-by-Pointer Application

2.3 Arrays as Pointers to Memory

One of the most common uses for pointers in C and C++ is when we are working with arrays. Arrays are just blocks of memory, and as pointers hold memory locations, an array is just a pointer to the starting memory location of the block of memory representing the array. This allows us to pass around large blocks of memory as a pointer thus avoiding copying.

Arrays Revisited

We have covered arrays in C in a few places now, although we haven't delved to far into the subject (this will happen in the next unit). Remember to create an array in C we use the following:

type `name[size];`

This will create a *fixed* (compile time defined) size array. This is an important distinction in C in comparison to Java. In Java, array sizes are defined at runtime as standard. In C and our arrays are of fixed size at compile time unless we use memory allocation.

In C, memory allocation is handled using the `malloc` function. This returns a pointer to a memory block of the required size.

In C arrays are a pointer to the start of the memory block where the memory is located. The size of the array is not stored however (unlike in Java) so we have to pass this value around. This is why working with `vector` in C++ (later in the unit) is almost always a better choice.

An example array application is below:

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

int* create_array()
{
    // This memory is created on the stack
    int data[20];
    for (int i = 0; i < 20; ++i)
        data[i] = i;
    return data;
    // Stack emptied - memory gone
}

int* create_array_new()
{
    // Memory created on the heap
    int *data = (int*)malloc(sizeof(int) * 20);
    for (int i = 0; i < 20; ++i)
        data[i] = i;
    return data;
    // Memory on the heap still relevant
}

void create_array(int *data)
{
    for (int i = 0; i < 20; ++i)
        data[i] = i;
}

int main(int argc, char **argv)
{
    int *data = create_array();
    // Print out all elements
    for (int i = 0; i < 20; ++i)
        printf("%d\n", data[i]);

    data = create_array_new();
    // Print out all elements
    for (int i = 0; i < 20; ++i)
        printf("%d\n", data[i]);
    // Free the memory
    free(data);
    // Set to NULL
    data = NULL;

    // Create array from pointer
    // This will cause a memory allocation error
    // NULL (memory address 0) cannot be allocated to
    create_array(data);

    // Allocate memory
    data = (int*)malloc(sizeof(int) * 20);
```

```
53      // Create array from pointer
54      create_array(data);
55      // Print out all elements
56      for (int i = 0; i < 20; ++i)
57          printf("%d\n", data[i]);
58      // Free the memory
59      free(data);
60      // Set to NULL
61      data = NULL;
62
63      return 0;
64  }
```

Listing 2.5: Passing Arrays as Pointers

Running this version of the code will cause a runtime error (segmentation fault) because of line 48 (or more specifically on line 26 which is called by line 48). Try running the application as is to check this. After that, comment out line 48 and run. You should get an output as follows:

```
0
14754591
14908184
14908272
6291172
14908272
6291164
14754917
6291172
6291184
14768466
14908272
-255
14908272
6291192
14751355
6291208
14755646
14908272
14908272
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
0
1
2
3
4
5
6
7
8
9
10
11
```

```
12
13
14
15
16
17
18
19
```

Listing 2.6: Output from Arrays as Pointers Application

The interesting values printed out are the first 20. This is what happens when we return a pointer to a location on the stack which is no longer valid (the GNU compiler actually warns us about this). The data here has been overwritten. This means that we have a variable that is pointing to an area of the stack we may use later, which will cause a possible *stack corruption* problem later. Stack corruption is where our stack is modified unintentionally, commonly by pointing to an invalid location on the stack. Figure 2.2 provides an illustration of what is happening.

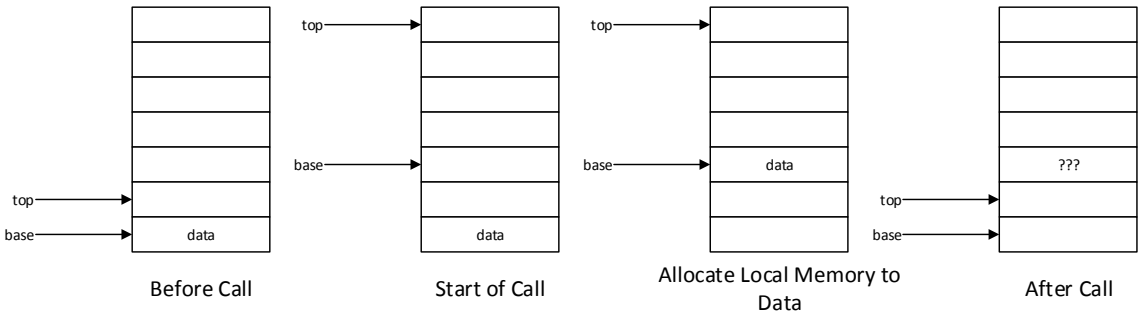

Figure 2.2: Data and Stack Corruption Example - Returning Non-Valid Memory

2.4 `const` Pointers and Pointers to `const`

Now let us look at how `const` affects pointers. There are actually two potential `const` parts to a pointer - the data stored in the pointer (*constant data*) and the memory location pointed to by the pointer (*constant pointer*). Table 2.2 illustrates the different approaches.

`const` Part	Example	Description
Value	`const int *x`	The value pointed to by the pointer is constant and cannot be changed. The memory address can be changed however.
Pointer	`int *const x`	The memory address cannot be changed (is constant). However, the value pointed to can be changed.
Value and Pointer	`const int *const x`	Both the value pointed to and the memory address are `const`.

Table 2.2: `const`ness of Pointers

Let us now build an example application. The code below has a number of lines you will have to comment out, but you should first build the code without the commented out lines. This is so you can see the compiler error that C will provide.

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

void foo(const int *x)
{
    printf("Address of x in function = %p\n", (void*)x);
    printf("Start of function, x = %d\n", *x);
    // Wont compile - value pointed to is const
    *x = 20;
    printf("End of function, x = %d\n", *x);
    printf("Address of x at end of function = %p\n", (void*)x);
}

void foo2(int *const x)
{
    printf("Address of x in function = %p\n", (void*)x);
    printf("Start of function, x = %d\n", *x);
    // Will compile - pointer is const, not value pointed to
    *x = 20;
    printf("End of function, x = %d\n", *x);
    printf("Address of x at end of function = %p\n", (void*)x);
}

void foo3(int *const x)
{
    printf("Address of x in function = %p\n", (void*)x);
    printf("Start of function, x = %d\n", *x);
    // Won't compile - trying to change address pointed to
    x = NULL;
    printf("End of function, x = %d\n", *x);
    printf("Address of x at end of function = %p\n", (void*)x);
}

int main(int argc, char **argv)
{
    int x = 10;
    printf("Starting address of x = %p\n", (void*)&x);
    printf("Before function call, x = %d\n", x);
    // Have to pass the pointer (or address of) x to the function
    foo2(&x);
    printf("After function call, x = %d\n", x);
    printf("End address of x = %p\n", (void*)&x);

    return 0;
}
```

Listing 2.7: `const` Pointers and Pointers to `const`

The compiler error provided by the GNU C compiler is as follows:

```
const_pointer.c: In function   void foo(const int*)   :
const_pointer.c:9:8: error: assignment of read-only location    * x
     *x = 20;
        ^
const_pointer.c: In function   void foo3(int*)   :
const_pointer.c:29:7: error: assignment of read-only parameter    x
     x = NULL;
       ^
```

Listing 2.8: Compiler Error from Trying to Modify a `const`

What is NULL?

We have introduced another new term in this code - NULL. This provides the equivalent of null in Java. NULL effectively points at memory location 0. NULL is a commonly defined value in C and C++ and is also 0. It is quite common to point unallocated memory to location 0 to allow code checks.

We've only begun to stray into pointers here, and the next unit will look at memory allocation and management. Here we have to use pointers, although the C++11 standard has introduced some helper objects to make our life easier.

Unit 3

Memory Management - Using the Stack and Heap

In this unit we are going to expand on our work with understanding pointers by beginning to work with memory. When working with a low-level systems language such as C (and C++), working with memory is an important consideration. So far, our journey through C has been as follows:

1. Introduction to programming in C, learning how our code is compiled and linked.
2. Learning how our data is represented in the computer's memory.
3. Learning how our C code is converted into Assembly language (and therefore machine understandable instructions).
4. Learning how our code files are processed and joined together to build compilation units.
5. Learning how we can pass values into functions using pointers (and references if you did C++).

In this unit we are going to investigate how we allocate memory, the difference between the stack and the heap, as well as looking a bit deeper into scope. For those of you doing well, we will also look at the new C++ features called *smart pointers*. For those of you who are a little more adventurous we will also look at basic value casting in C++.

3.1 Scope of Values

We have already looked into scope in a basic manner. In this part we will look at scope in a bit more detail. In particular, we will build inner scopes in our functions to give you an idea of how scope works in a bit more detail.

What is Scope?

Scope is about which values are currently valid in a particular piece of code. In general, you should understand scope from the basic idea of having a variable available only after you have declared it. For example:

```
x = x + 1; // Don't know what x is here
int x = 0; // x only declared now
```

As `x` is declared after we use it we will get a compiler error - `x` is undeclared (not in scope).

Scope becomes more complex when we work with functions and classes. When we call a function, we pass any variables we want to the function's scope. In the previous unit we looked at how this could mean passing by value or passing by reference. If we don't pass in the variable, then the value is not in scope.

Scope actually works in C / C++ from the point of view of values declared within braces. For example, consider the following:

```
void func()
{
    // Main scope of the function
    {
        // Scope A - can see main scope
        {
            // Scope B - can see scope A and main scope
        }

        {
            // Scope C - can see scope A and main scope.  Scope
               B no longer valid
        }
    }
}
```

We can only see values in our outer scope - not our inner scope. This means whenever you use a curly brace (such as in a `while` loop or `if` statement) you create a new inner scope. Any values declared in these scopes are destroyed (removed from the stack) when the scope is exited.

In C and C++ we have the ability to create values that are in the *global* scope. This can be useful, but is often frowned upon. Passing values around the application as parameters is considered best practice.

Scope can be a tricky concept for new programmers. Spend your time understanding which values are valid at particular points of your application. Our next two example applications explore scope in more detail.

3.1.1 Which x is in Scope?

Our first application investigating scope will look at how we can declare new values in inner scope with the same variable name, but still retain the values in the outer scope. In a way, you can consider the inner scopes as the scope of a function (without parameter passing). Below is our example application. Notice how we keep redeclaring `x` in each inner scope, then unwind to get back to the original value.

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

int main(int argc, char **argv)
{
    // Outermost declaration
```

```
7       int x = 10;
8       printf("Outermost x = %d\n", x);
9       {
10          // Now in new scope
11          int x = 20;
12          printf("\tInner x = %d\n", x);
13          {
14              // Now even further in scope
15              int x = 30;
16              printf("\t\tInner inner x = %d\n", x);
17              {
18                  // Let's stop here
19                  int x = 40;
20                  printf("\t\t\tInnermost x = %d\n", x);
21                  // Now unwind scope
22              }
23              printf("\t\tInner inner x = %d\n", x);
24          }
25          printf("\tInner x = %d\n", x);
26      }
27      printf("Outermost x = %d\n", x);
28
29      return 0;
30  }
```

Listing 3.1: Multiple `x` Values in Different Scopes

Each time we enter a new scope we redeclare our `x` variable. *We are not redefining the value stored in* ***x*** *- we are creating a new variable.* This is an important concept to understand. As these are new variables, we are not changing the value of the previous scope. It still exists. Therefore, this application will print out values of each scope, then return back to the first scope. The output from this application is below:

```
Outermost x = 10
        Inner x = 20
                Inner inner x = 30
                        Innermost x = 40
                Inner inner x = 30
        Inner x = 20
Outermost x = 10
```

Listing 3.2: Output from Scope Test Application

3.1.2 Values out of Scope

So what happens when a value is out of scope? Well we end up in a situation where we have *undeclared identifiers*. This leads to a compiler error. Effectively the value hasn't been declared from the point of view of the compiler and therefore cannot be used. The following application illustrates this:

```
1   #include <stblib.h>
2   #include <stdio.h>
3
4   int main(int argc, char **argv)
5   {
6       // Declare an int here
7       int i = 10;
8       {
9           // Declare another int here - can access i
10          int j = i * 2;
```

```
11          {
12              // Declare another int here - can access i and j
13              int k = i + j;
14          }
15          // Compile error here - k not in scope
16          j = j + k;
17      }
18      // Compile error here - j not in scope
19      i = i + j;
20
21      return 0;
22  }
```

Listing 3.3: Trying to Access Out of Scope Values

We have two problems in this code. On line 15 we are attempting to access variable `k`, but this was declared in a scope that we have now exited (it existed in lines 10 to 14). Therefore the compiler will give an error. A similar problem exists on line 18. `j` was only in scope lines 7 to 16. At line 18 it no longer exists (it has been removed from the stack) and therefore our compiler again throws an error. Compiling this application provides the output below:

```
scope2.c: In function   int main(int, char**)   :
scope2.c:16:17: error:    k    was not declared in this scope
         j = j + k;
                 ^
scope2.c:19:13: error:    j    was not declared in this scope
     i = i + j;
             ^
```

Listing 3.4: Compiler Output from Out of Scope Variable Application

Those errors from the compiler are important to spot. The fact that values are not in scope is given and the particular line the problem exists in provided. Being able to spot problems with out-of-scope variables is another stumbling block for new programmers.

3.1.3 Losing Values on the Stack

So now we know the pitfalls of working with scope we can relate back to what we have been working to up until this point. In the last unit we discussed passing values as values, references and pointers. In this unit we are interested in memory management. Our problem really comes into effect when we allocate a block of data on the stack and then try and return it. Remember that when we return from a function we lose the stack created for the function.

The problem becomes apparent when working with arrays. As we have seen in C and C++ we can declare an array on the stack in our code if we know the size of our array at compile time.

```
int array[10];
```

If we do this in a function the data declared on the stack is lost when we exit its scope. Therefore, if we return a such an array, the pointer is no longer pointing to valid memory. This will happen also if we try to return a pointer to a value declared in a function. We are effectively pointing to a location on the stack that has been deemed no longer allocated.

We get around this by allocating memory outside the stack - in the global memory space or heap. Before discussing the heap in any detail let us look at an example

application that shows what happens when we return values from a function using a pointer. We also illustrate what happens when we try to set values in a memory location we are not allowed to.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int* create_array()
5  {
6      // This memory is created on the stack
7      int data[20];
8      for (int i = 0; i < 20; ++i)
9          data[i] = i;
10     return data;
11     // Stack emptied - memory gone
12 }
13
14 int* create_array_new()
15 {
16     // Memory created on the heap
17     int *data = (int*)malloc(sizeof(int) * 20);
18     for (int i = 0; i < 20; ++i)
19         data[i] = i;
20     return data;
21     // Memory on the heap still relevant
22 }
23
24 void create_array(int *data)
25 {
26     for (int i = 0; i < 20; ++i)
27         data[i] = i;
28 }
29
30 int main(int argc, char **argv)
31 {
32     int *data = create_array();
33     // Print out all elements
34     printf("Array 1\n");
35     for (int i = 0; i < 20; ++i)
36         printf("%d\n", data[i]);
37
38     data = create_array_new();
39     // Print out all elements
40     printf("\nArray 2\n");
41     for (int i = 0; i < 20; ++i)
42         printf("%d\n", data[i]);
43     // Free the memory
44     free(data);
45     // Set to NULL
46     data = NULL;
47
48     // Create array from pointer
49     // This will cause a memory allocation error
50     create_array(data);
51
52     // Allocate memory
53     data = (int*)malloc(sizeof(int) * 20);
54     // Create array from pointer
55     create_array(data);
56     // Print out all elements
57     printf("\nArray 3\n");
```

```
58      for (int i = 0; i < 20; ++i)
59          printf("%d\n", data[i]);
60      // Free the memory
61      free(data);
62      // Set to NULL
63      data = NULL;
64
65      return 0;
66  }
```

Listing 3.5: Trying to Access Out of Scope Variables on the Stack

If you try and run this application you will get a runtime error (the application will hang) because of the attempt to allocate to `NULL` (which represents memory location 0). If you fix the code and run the application you will get the following output for Array 1.

```
Array 1
16741608
16590431
16747288
16747376
10157064
16747376
10157056
16590789
10157064
10157076
16604322
16747376
-255
16747376
10157084
16587195
10157100
16591518
16747376
16747376
```

Listing 3.6: Array 1 from Function Return

Notice that these are not the values you are expecting. The array has been cleared and now we have random values in the memory.

3.2 Allocating Data in Global Memory (the Heap)

So we now know the limitation of working with the stack when trying to return values from functions. Overcoming this involves us working with global memory. Global memory (the heap) allows us allocate memory that is not freed until such time as we wish to free it. This is advantageous for a number of reasons, but also leads us to some of the biggest problems for new *and experienced* software developers. There is a reason why Java hides this with a garbage collector. However, understanding memory allocation can be very important to comprehend when your values are valid and when they are cleared up.

In a way, you can consider the stack as working memory - it is very ordered, is cleaned up when finished with, and is also quite limited (more on this later). The heap is a large blob of memory that we can allocate to and keep values on for long term storage. Figure 3.1 tries to illustrate this idea.

The question now is how do we allocate memory on the heap? Well we have already seen this in a few places through the previous few units. Now let us look at this in more detail, starting with how we do this in C for raw blocks of memory.

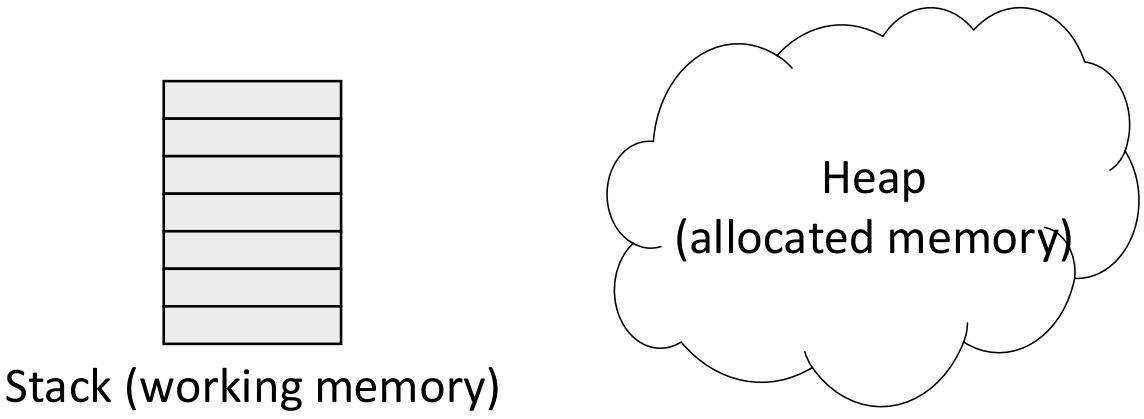

Figure 3.1: Stack and Heap

3.2.1 The `malloc` Function

We have already seen the `malloc` function when we worked with file I/O. `malloc` stands for Memory ALLOCation. It is used to allocate a block of memory of a given size on the heap. It takes the following form:

```
void *variable = malloc(size);
```

Notice first that `malloc` returns a pointer to `void`. This means that the type of memory isn't defined. `malloc` simply returns a pointer to a block of memory. This means that you have to cast it to the relevant type (e.g. `int*`).

The only parameter that `malloc` takes is the number of bytes that need to be allocated. This is the `size` parameter. Remember that this is the number of *bytes* being allocated. Typically we use `sizeof` and the number of values we want to determine the number of bytes that we need. For example, to allocate 100 `int` values on the heap we would use:

```
int *data = (int*)malloc(sizeof(int) * 100);
```

This is all we need to know about allocating memory in C. For those wanting to know more, you can look into C++ allocation shortly. However, first we need to look at how we deallocate memory.

3.2.2 The `free` Function

In C and C++ we are usually responsible for allocating memory and ensuring that it is cleaned up afterwards. In C, this involves us using the `free` function.

```
free(value);
```

`free` takes the pointer created by a call to `malloc`. It doesn't need to know the size of the data as this is kept track of. Essentially your calls should look something like this:

```
1 int *data = (int*)malloc(sizeof(int) * 100);
2 // Do some work with data.  Once finished call free
3 free(data);
```

Why do we need to `free`?

One of the biggest disadvantages levelled at C and C++ is the fact that you have to be responsible for tracking memory that has been allocated on the heap. Once you have finished with it, you call `free` to deallocate. If you don't free used memory, then your application slowly increases its memory usage until it runs out.

A bigger problem occurs when lose a pointer to allocated memory by it going out of scope. This is a *memory leak*. Without the pointer, you cannot free the memory. Do this enough times, and the above problem occurs, but this time we have no way to rectify the issue. This is one of the issues new C/C++ programmers commonly face as well.

3.2.3 Using `calloc` to Clear Memory While Allocating

`malloc` is not the only method in C to allocate memory. The other method is using `calloc`. This function works by not only allocating memory but also zeroing the memory allocated (all locations have 0 stored). This takes a bit longer than `malloc` but does ensure no previous values are stored in memory.

The call to `calloc` is a little different than `malloc`:

```
void *data = calloc(number of values, size of values);
```

Notice that the `size` parameter from `malloc` has been split into two values - the number of values you require and the size of those values. Essentially it is the same calculation as before, just split across two values.

As with `malloc` any call to `calloc` has to have its memory deallocated using `free`.

3.2.4 Example

With the necessary calls for working with memory in C discussed we can now build an example application. The following uses three methods for creating and returning an array of data.

1. On the stack (bad)
2. Using `malloc`
3. Using `calloc`

The example application is below.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int* foo()
5  {
6      // This memory is created on the stack
7      int data[20];
8      for (int i = 0; i < 20; ++i)
9          data[i] = i;
10     return data;
11     // Stack emptied - memory gone
```

```
12 }
13
14 int* foo2()
15 {
16     // Memory created on the heap
17     int *data = (int*)malloc(sizeof(int) * 20);
18     for (int i = 0; i < 20; ++i)
19         data[i] = i;
20     return data;
21     // Memory on the heap still relevant
22 }
23
24 int* foo3()
25 {
26     // Memory allocated on the heap using calloc
27     // Memory set to 0.  Use elements and element size as seperate
       parameters
28     int *data = (int*)calloc(20, sizeof(int));
29     return data;
30     // Memory on the heap still relevant
31 }
32
33 int main(int argc, char **argv)
34 {
35     // Call foo - stack is corrupted
36     int *data = foo();
37     // Check values
38     printf("Array 1\n");
39     for (int i = 0; i < 20; ++i)
40         printf("%d\n", data[i]);
41
42     // Call foo2 - data on heap so not lost
43     data = foo2();
44     printf("Array 2\n");
45     for (int i = 0; i < 20; ++i)
46         printf("%d\n", data[i]);
47     // Free memory
48     free(data);
49
50     // Call foo3 - data on heap so not lost
51     data = foo3();
52     printf("Array 3\n");
53     for (int i = 0; i < 20; ++i)
54         printf("%d\n", data[i]);
55     // Free memory
56     free(data);
57
58     return 0;
59 }
```

Listing 3.7: Allocating and Freeing Memory on the Heap

The output generated from this application is as follows:

```
Array 1
8994208
8920684
17
10090664
8917873
1
8987032
8917853
1181712074
```

```
0
0
0
2
10090620
10090624
10090752
8924336
1182763074
-2
8917853
Array 2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
Array 3
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
```

Listing 3.8: Output from Allocating Memory on the Heap Application

As we can see, Array 1 has corrupted values. Array 2, which is allocated on the heap, is correct. Finally, for `calloc` we can see that the numbers are zeroed in Array 3.

3.3 Limits of the Stack

So far we have seen that using the heap can overcome values being out of scope. However, we already found out in the last unit that we can overcome this by passing in pointers and references. We could *possibly* get around many of our problems in this way, but actually there is a particular issue that we have to deal with - *the limit of the stack*.

Your applications only have a limited amount of stack space. The general default on Linux is 7.4 megabytes. This isn't that much really. This means that when you

are working with any application that needs more that 7.4 megabytes of stack space you need to use the heap. In any reasonably sized application this will be the case.

To test the limits of the stack, try the following application:

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

// Size of data to allocate
const unsigned int SIZE = 8388608;

int main(int argc, char **argv)
{
    // Try and create data on the stack
    // This is 8MB - greater than common stack size
    char data[SIZE];
    for (int i = 0; i < SIZE; ++i)
        printf("%d\n", data[i]);

    return 0;
}
```

Listing 3.9: Allocating Beyond the Limit of the Stack

When you compile and run this application you will get a runtime error (segmentation fault). This is because we have attempted to allocate an array of 8 megabytes on the stack.

3.4 Allocating Large Memory Blocks on the Heap

So the heap is the go to area of memory to work with shared memory and large data blocks. However, there is a limit to how memory you can allocate on the heap. The following test application lets us create a large 1 gigabyte area of memory. This is fine, but work on the exercise to see how far you can push the limits of the application.

```
#include <cstdlib>
#include <cstdio>

// 1 megabyte in bytes
const unsigned int MB = 1024 * 1024;
// 1 gigabyte in bytes
const unsigned int GB = 1024 * MB;

int main(int argc, char **argv)
{
  // Allocate 1 GB of data
  char *data = (char*)malloc(GB);

  // Need to "use" the data before actual allocation occurs
  for (int i = 0; i < 100; ++i)
    printf("%d\n", data[i]);

    // Free the memory
    free(data);

    return 0;
}
```

Listing 3.10: Allocating 4GB of Memory with `malloc`

We use `malloc` here to allocate 1 gigabyte of data. Many operating systems and compilers will optimise your application so that memory isn't actually allocated until we use it. Therefore, we print out 100 values from our memory block to force the memory to be allocated. Running this application will cause no issues, but you should move onto the exercise to test some limits.

3.4.1 Exercise

Try and find the limit of the amount of memory you can allocate on 32-bit applications. There is a limit to the size a single variable can have (2 gigabytes), so create 1 gigabyte variables until you hit a problem. You should also allocate a 2 gigabyte variable just to see that this is an issue. *Do not try this with a 64-bit application!*. The limit on 64-bit will be larger than your memory and hard drive will likely be able to handle. The operating system will allocate memory into your hard drive swap space if there is not enough main memory to cope. If you are running a 64-bit OS you will need to install `g++-multilib`.

3.5 Using Pointers to Pointers to Allocate Memory to Parameters

So we have now worked on the boundaries of using memory allocation. There are still another problem that new C and C++ programmers experience - allocating memory to parameters passed into a function. Remember in the last lesson that we used pointers and references to pass in values to functions that we could then change. Well a pointer itself is just a value, so what if we want to change the pointer (i.e. the memory location pointed to) in a function. This is where we need pointers to pointers.

What are *Pointers-to-Pointers*?

A pointer to a pointer is exactly that. We have a memory location that itself contains a memory location telling us where the data is actually located. We have two levels of indirection. At the moment, we need these ideas to allow us to allocate memory in a function to a passed in parameter. We will look at pointers-to-pointers further when we discuss data structures towards the end of the module. We have already used pointers-to-pointers when we worked with the command line (remember the `char **argv` value).

We can have many levels of indirection. It is possible to have a pointer-to-pointer-to-pointer - a 3-dimensional array will have this, or a 2-dimensional array passed in as a parameter. The point is we have memory locations that contain memory locations. We can then work with these to various levels. Although common in C, they are less common in C++ where we can use object-orientation to overcome the issues.

The following example application illustrates the problem when using `malloc` in a function. Of particular note is the memory leaks we create. Note that compiling this application will give you some warnings from `printf` but these can be ignored.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
```

```
4  void foo(int *value)
5  {
6      // Allocate the value.  This changes the memory pointed to
7      value = (int*)malloc(sizeof(int));
8      *value = 5;
9      // Print address and value
10     printf("In first foo, address - %d, value - %d\n", value, *
           value);
11     // When we return, we lose the memory address
12     // We have a leak!!!
13 }
14
15 void foo(int **value)
16 {
17     // Allocate the value.  This changes the memory pointed to
18     *value = (int*)malloc(sizeof(int));
19     **value = 5;
20     // Print address and value - have to dereference twice
21     printf("In second foo, address - %d, value - %d\n", *value, **
           value);
22     // When we return, we retain the pointed to memory address
23 }
24
25 int main(int argc, char **argv)
26 {
27     // Declare value.  Use calloc to set as 0
28     int *value = (int*)calloc(1, sizeof(int));
29     // Print address and value
30     printf("Initially, address - %d, value - %d\n", value, *value);
31     // Call foo
32     foo(value);
33     // Print address and value
34     printf("After first foo, address - %d, value - %d\n", value, *
           value);
35     // Call foo with pointer to pointer
36     foo(&value);
37     // Print address and value
38     printf("After second foo, address - %d, value - %d\n", value, *
           value);
39     // Free memory
40     free(value);
41
42     return 0;
43 }
```

Listing 3.11: Using Pointers to Pointers

Memory Leak Problems

Once we "lose" a memory address (the pointer) we cannot free the memory involved. However, the Operating System will still have the memory marked as allocated until your application exits. We have seen a few ways now that we can create memory leaks and memory allocation problems and for the novice programmer remembering these can take time.

In current software development using the types of systems we do losing a few bytes of memory in an applications lifetime can be seen as not a bad thing. A few years ago when memory was far more limited it was a bigger issue. On

limited systems (such as embedded systems) losing a few bytes of memory can be an issue.

In an application that has some form of control loop (such as game that loops 60+ times a second) losing a few bytes every iteration is a major issue as that memory won't come back until the application exits. This can escalate quickly to your application existing with a memory problem or just slowing down considerably.

Dereferencing Multiple Times

In the above application we also use `**` to dereference our pointer-to-pointer value. This is also worth remembering as a new programmer - *for every level of indirection you create you have to perform a dereference to get to the actual value*. When we work with multiple dimensional arrays later in the module we will come back to this idea in more depth.

Running this application will provide you with output similar to the following (the memory addresses will be different):

```
Initially, address - 16833728, value - 0
In first foo, address - 16833744, value - 5
After first foo, address - 16833728, value - 0
In second foo, address - 16833760, value - 5
After second foo, address - 16833760, value - 5
```

Listing 3.12: Output from Pointer to Pointer Application

Notice the problem that occurs in the first call to `foo`. We can see that we have allocated a new block of memory (location 16833744 above) and set the value to 5. However, outside the call this is not reflected - the memory location and value is the same as before the call. Only in the second `foo` do we solve the problem.