Edinburgh Napier University

SET07109 Programming Fundamentals

Workbook 2018/19

Created by Dr Kevin Chalmers
Maintained by Dr Simon Powers

Contents

Page

List of Figures

List of Algorithms

Unit 1

Compiling and Linking Programs

Welcome to the practical part of the Programming Fundamentals module. This module is designed to give you a strong underpinning in programming, looking at low level concerns such as how code is run on the machine to high level concerns such as an introduction to object-orientation. The goal of the module is enhance your programming skills. As such, you should work through all the examples and exercises that you can. Some of the work is for the really keen members of the class, but with perseverance everyone should be able to work through all the module content.

1.1 Getting Setup

This module takes a bottom up approach to the teaching of programming. As such, you will initially work using a text editor and compiling from the command line to understand the basic compilation process. Half way through the module we will shift into Microsoft Visual Studio to do our work.

The toolchain we are going to use is Microsoft's set of C/C++ compilers, linkers and assemblers. This is because we are using Visual Studio later in the module. However, you could use other C/C++ compilers such as `gcc` and `clang` if you are a Linux or Mac OS X user. The code we are using in the first part of the module is standard C/C++. In fact throughout the module we will almost always use standard C/C++. It is only when looking at application debugging do we really use Microsoft specific technology.

1.1.1 Downloading Notepad++

If you are using Microsoft Windows, the text editor we recommend for the module is Notepad++ (`http://notepad-plus-plus.org/`). Notepad++ is free, provides code highlighting, and can be run from a USB key. It is already installed on all of the University machines and is accessed through *Apps Anywhere*.

Other text editors exist. Windows comes with Notepad. Linux and Mac OS X come with a variety of tools. Sublime Text is a very popular text editor, but it isn't free. From the point of view of the module, the team will not specify a text editor you must use. It is up to your own personal preference.

We will look at Notepad++ as an example of starting up. When run, you should have a startup screen as shown in Figure 1.1.

In the *File* menu you can create and save files. *For code highlighting to work, you must save the file with the correct extension.* For the first few units we will

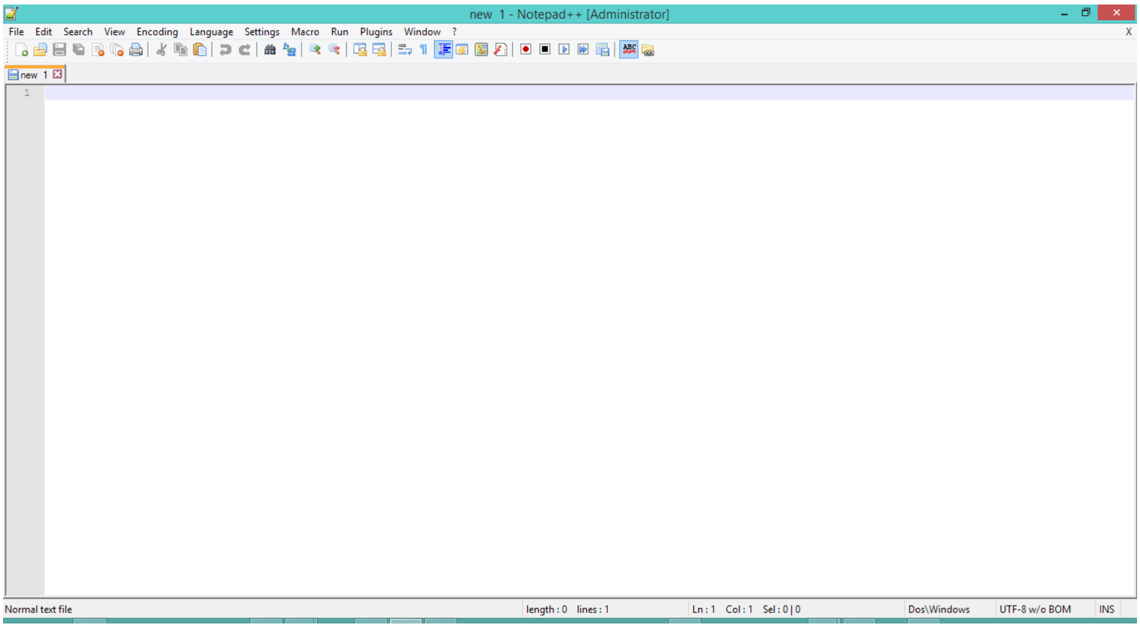


Figure 1.1: Notepad++ Start Screen

be using C. This means all files should be saved with the `.c` extension. It is also recommended that you save your files somewhere sensible. The `H` drive is probably the best place, although if you are familiar with version control, a repository is even better. DropBox, OneDrive, etc. are options, but can give you conflicts when we move onto Visual Studio.

1.1.2 Running from the Command Line

We will be working from the command line to compile and run our applications. This means you need to open the Visual Studio Developer Command Prompt. To do this, you first need to launch Visual Studio 2017 through *Apps Anywhere*. Once the Visual Studio main window is open, minimise it (we will not be coding in this window). Then go to the *Start Menu*, *All Programs*, *Visual Studio 2017*, *Visual Studio Tools*, and choose *Developer Command Prompt for VS 2017*.

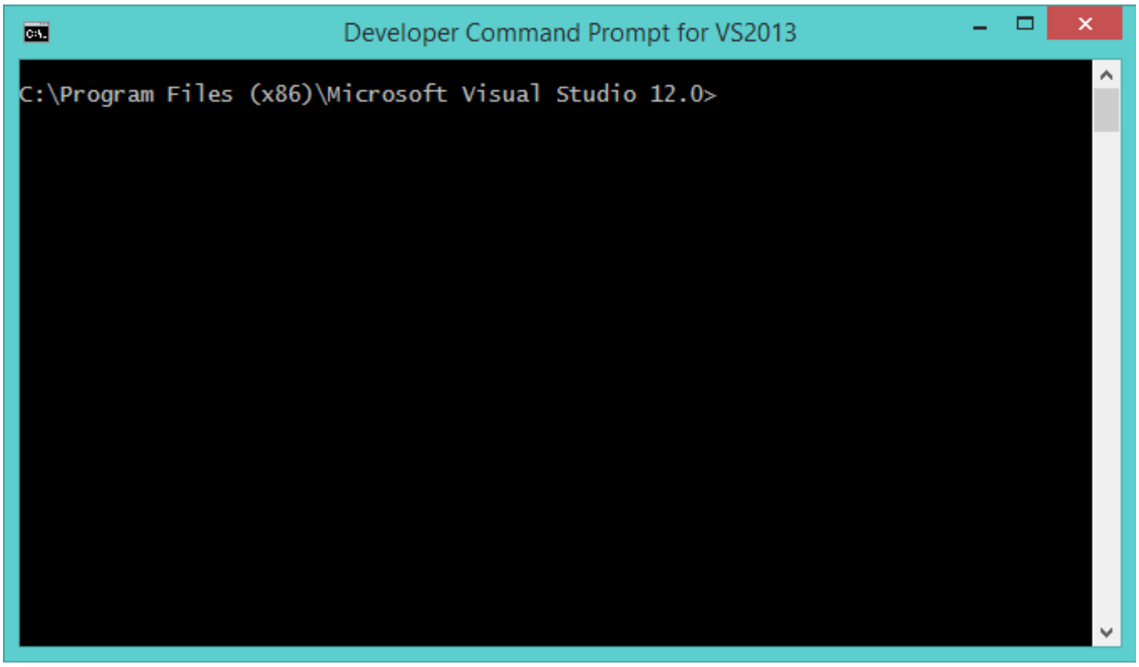
We now need to use some command line instructions to move to the correct folder where you are saving your files. Let us say that you are storing your files on your `H` drive under a folder called `set07109`. The commands to use are:

```
H:
cd set07109
```

The first command (`H:`) changes the drive being accessed to the `H` drive. The second command (`cd set07109`) changes the folder to `set07109`. If you need to go up a directory in the folder structure, you can use the `cd ..` command. You can find out more about using the command prompt at `http://dosprompt.info/`.

1.2 Hello World! What Else?

As with every introduction to a new programming language we will start with a simple Hello World application. However, before diving straight into that let us do a brief introduction to the C programming language.



```
C:\Program Files (x86)\Microsoft Visual Studio 12.0>
```

Figure 1.2: Visual Studio Command Prompt

1.2.1 The C Programming Language

C is one of the most (if not the most) popular programming languages in the world. It is also one of the oldest languages still in major use (C was developed between 1969 and 1973). Unlike modern languages (e.g. Java), C is not object-oriented. It is quite a simple language, and is very close to how the CPU operates (C is often called a *systems programming language*). Most operating systems are written in C, and most operating system APIs are C based.

C is considered the base language of many other common languages. Many languages are referred to as *C like*. These include Java, C#, JavaScript, Python and PHP. As such, the syntax of C is familiar to nearly all programmers as they will generally be working in a C based language. It could be argued that nearly all commercial software in the world is programmed by *C like* programmers.

We will explore C as we work through the module. There are some differences when compared to Java and other modern languages. In particular, C relies on an understanding of memory and memory management to fully appreciate. We will work towards this understanding as we work through the module.

1.2.2 Hello World Code

OK, let us dive right in and write our first C application. As stated, we will write a Hello World application. This code should be saved in a file called `hello.c`. The code is given below.

```c
// stdio contains standard C input-output functions
#include <stdio.h>

// Our main function.
// argc is the number of command line arguments
// argv are the command line arguments
// We will look at these later
int main(int argc, char **argv)
{
```

```
10     // Print hello world to the screen
11     printf("Hello world!\n");
12     // Return value from program
13     return 0;
14 }
```

Listing 1.1: Hello World in C

Let us look through this code to get an idea of what is going on:

- Lines starting with a // are comments. These lines are ignored by the compiler. You should use comments as much as possible to understand your code.

- Line 2 is `#include <stdio.h>`. Lines that start with a # are pre-processor commands. We will look more at the pre-processor in the next unit. This line is including the functionality in the `stdio.h` file. This is part of the C standard library, and allows us to perform basic input-output (I/O) commands. Hence the name `std` (standard) `io` (input-output). Including existing code makes our life much easier.

- Line 8 is the start of our main application. The C compiler expects the main part to have the name `main`. We also define two incoming parameters - `argc` and `argv`. We will look at these values later in this unit.

- Line 11 is where we print out our Hello World message. To do this we use the `printf` function. Also note the use of `\n`. This denotes that a new line should be added to the end of the printed line.

- Line 13 is where we exit the application by returning from our `main` function. This returned value can be used by other applications to determine the successful completion of our Hello World application. In general, 0 is considered OK and other values are considered error codes. We won't really look into this.

`printf`

The `printf` function is used to print text to the command line. It takes the following form:

`printf("format", params...);`

The `format` part of the call is the most interesting. We will look at some examples of this through the module. However, the following examples should help:

- `printf("Hello World\n");` - prints 'Hello World' to the command line

- `printf("x = %d \n", x);` - prints the value of `x`. For example, if `x` is 5, then it prints 'x = 5'. The `%d` value is a place holder telling `printf` to insert the parameter in it's place

- `printf("Hello %s. You are %d years old \n", name, age);` - prints a welcome message. For example, if `name` is Kevin and `age` is 36 then it prints 'Hello Kevin. You are 36 years old'. `%s` is another place holder. Notice that values are placed in the place holders in the order they are given as parameters.

The place holders we will look at through the next few units. However, the two we have used just now are:

`%d` an integer value

`%s` a string of characters

Escape Characters

The `\n` value in the `printf` statement is what we call an *escape character*. This is a character we cannot type simply using a keyboard. The `\n` value adds a new line character. Other escape characters we will come across are:

- `\t` - inserts a tab
- `\0` - null terminator. Used to end strings of characters

Wikipedia provides a comprehensive list of escape characters - `http://en.wikipedia.org/wiki/Escape_sequences_in_C`.

Code structure

Code quality is one of the most important habits you should pick up as a new programmer! Having tidy, well laid out code will make your life easier in the long run. Employers want to see your code, and will expect it to be of high quality. You will also be assessed in university on the quality of your code.

For this module you need to adopt a style and stick to it. In C/C++, the standard style is to have curly brackets on new lines. It is likely you have seen Java and JavaScript open curly brackets on the same line. If this is what you prefer, OK. But be consistent.

Indentation is also important. Code blocks should be indented based on scope. This is fundamental and carries across all languages. Get in the habit now.

1.2.3 Building your Application

We are now ready to compile our application. To do this we are using Microsoft's C/C++ compiler - `cl`. We will visit some of the different commands we can give `cl` through the module. For the moment we will perform the simplest task - having `cl` build an executable from a single `.c` file. The command to do this is as follows:

```
cl hello.c
```

Listing 1.2: Building our Hello World Application

Once run, you should receive the following output:

```
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.30501 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

hello.c
Microsoft (R) Incremental Linker Version 12.00.30501.0
Copyright (C) Microsoft Corporation.  All rights reserved.
```

```
/out:hello.exe
hello.obj
```

Listing 1.3: Output from Building our Hello World Application

The compiler goes through two stages:

1. Compiling the code in `hello.c`

2. Linking the generated object file to the system libraries to allow running. We will look at this stage later in the unit.

If there is a problem, then the compiler will give you an error message. For example, if we remove the ; at the end of line 11 as follows:

```
 1 // stdio contains standard C input-output functions
 2 #include <stdio.h>
 3 
 4 // Our main function.
 5 // argc is the number of command line arguments
 6 // argv are the command line arguments
 7 // We will look at these later
 8 int main(int argc, char **argv)
 9 {
10   // Print hello world to the screen
11   printf("Hello world!\n")
12   // Return value from program
13   return 0;
14 }
```

Listing 1.4: Incorrect Hello World in C

when compiled the compiler will give the following output:

```
hello.c
hello.c(13) : error C2143: syntax error : missing ';' before 'return'
```

Listing 1.5: Error message returned from a build

Here, the error message says there is a problem on line 13 of hello.c. It is saying that there is a missing ; before this line. This is the ; we removed from line 11.

This is an important concept to understand. The compiler tries its best to tell you the line that has a problem, but sometimes it will tell you the line of code after. This is because the error doesn't cause a problem on the line we removed the ; from, but on the next one.

Make sure that your application compiles and then run it by typing `hello` at the command prompt:

```
hello
Hello world!
```

Listing 1.6: Running the Hello World Application

1.3 C Strings

So we've printed a message. Let us now look in more detail at how C treats strings of characters (how we store text). Being a low level language, C treats text just as it is represented in memory - just a block of memory with character data in it. The text has a starting point in memory (represented by a memory address or pointer), and C will treat every character in memory after this point as a member of the string until it finds a null terminator (\0 in text or 0 as a value). Figure 1.3 illustrates the general idea.

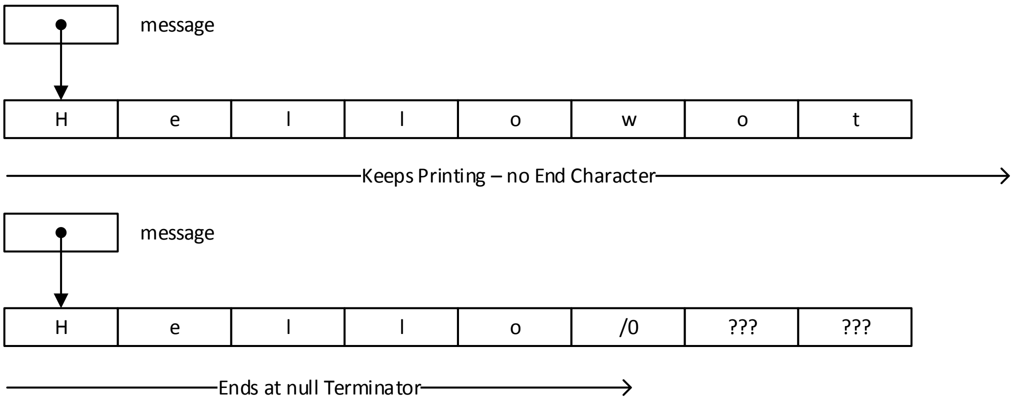


Figure 1.3: C Strings and null Termination

The reason C does this is because it takes up the least amount of memory overhead. Remember C is an old language. Back then, every byte of memory was precious. Using a null terminator for strings of characters had only a single byte overhead. A string retaining its size (as Java does) has a two or more byte overhead. Therefore, null terminators were used.

C provides a number of helper functions for working with strings. These functions are provided in the `string.h` header file. We will use some of these functions over the next few examples. Let us first look at different methods of declaring strings of characters in C.

```c
#include <stdio.h>
// contains helper functions for strings
#include <string.h>

int main(int argc, char **argv)
{
    // Declare a character array for message
    char msg_1[5] = {'H','e','l','l','o'};
    // Declare a second character array for message
    char msg_2[8] = " World!";
    // Declare a third message
    char msg_3[9] = "Goodbye!\0";
    // Declare a forth message - no size
    char msg_4[] = "Compiler worked out my size!";
    // Declare a fifth message - use a pointer
    char *msg_5 = "Compiler worked out my size too!";

    // Print messages.  1st message is not null-terminated
    printf("%s\n", msg_1);
    printf("%s\n", msg_2);
    printf("%s\n", msg_3);
    printf("%s\n", msg_4);
    printf("%s\n", msg_5);

    return 0;
}
```

Listing 1.7: C Strings

We have used 5 different techniques to declare a string of characters:

1. Declaring an array of `char` by declaring each individual member of the array. The array size of 5 means that there is no null terminator
2. Declaring an array of `char` by using a string initializer (using quotation marks). The string initializer automatically adds a null terminator. As the array is 8 characters long, then the null terminator is included.
3. Declaring an array of `char` using the string initializer and explicitly adding the null terminator. This actually defines two null terminators - the one we have and one after the string.
4. Declaring an array of `char` using the string initializer but not setting the size of the array. The compiler will work this out for us.
5. Declaring a pointer to a block of `char` data in memory. The compiler takes care of initialising the memory for us. There is actually no real difference between this method and the previous one. An array is just a block of memory.

Arrays in C

You have probably come across arrays by now, so this might be a bit of a refresh.

Arrays in C operate like other languages. We have the type of the array (for strings we are using character data or `char`) and the size (the number in the brackets). C requires that you know the size of the array at compile time. That is, the following code will not compile:

```
int x = 42;
char data[x]; // x is not a constant value
```

This is because C needs to allocate the required memory when the program is compiled. We will cover memory allocation in far more detail later in the module. At present, it is enough to know the standard format of array declaration:

`*type* name[size];`

Where `size` is a constant, compile time value.

What is a Pointer?

We've introduced the concept of pointers, and this is one of the areas new programmers in C get really caught up. We will be introducing pointers more and more throughout the module, and your understanding will grow over this time. At the moment, consider a pointer at its most basic form - it *points* to a location in memory, and tells the compiler the *type* of memory that is located there. A pointer effectively contains a memory address (think back to computer systems and memory here) which we can use to access a particular area of memory. This is how arrays are actually dealt with.

From the point of view of `printf` and strings, we pass `printf` a location in memory and tell it to consider this as character data. `printf` then goes through the memory, displaying the data on the screen using ASCII to convert values into characters, until such time it finds a null terminator (0).

Pointers are declared using the * type:

`*type* *name;`

> The use of * can become confusing as we will see it used to also *dereference* pointers, as well as the obvious use for multiplication. We will introduce these ideas slowly.

You should now compile and run your program. Let us assume you have saved the code in a file called `using-strings.c`. Then you would compile and run using the following commands:

- `cl using-strings.c`
- `using-strings`

If the program doesn't compile, try and fix your errors. If you are still having problems, ask for help.

Running the application should give you an output similar to the following:

```
Hello
 World!
Goodbye!
Compiler worked out my size!
Compiler worked out my size too!
```

Listing 1.8: Output from Strings Application

There is a very good chance that you will have some extra characters printed after the 'Hello' message. This is due to the lack of null termination. `printf` keeps printing characters in memory until it encounters a null terminator. This problem is actually a classic bug in low level programming, and can cause all sorts of problems with security.

1.4 Reading your Name

Outputting to the command line is one thing, but what about inputting messages. Let us look at how we capture input from the command line.

C actually provides a few different techniques for capturing input, but we are going to use the simplest one. The function we are going to use is called `fgets`. It takes the following form:

`fgets(`*`string`*`, `*`length`*`, `*`stream`*`);`

The parameters are as follows:

`string` - the array of memory (`char` array or pointer) to read into

`length` - the maximum number of characters to read. *This must be equal to or less than the size of memory being read into.*

`stream` - the location to read data from. This is a `FILE*` (pointer to a `FILE`) type. We will look at file input/output later in the module. However, we can treat the command line as a file. It has the name `stdin` (STanDard INput).

Let us create an example application for reading in a user's name from the keyboard. This example also uses some additional string functions which we will explain shortly. The application asks for a first name and last name, and then joins them together into a full name. Algorithm 1 provides the steps being taken.

The (almost complete) code listing for this application is given below:

Algorithm 1 Prompting for and Displaying the Username

```
procedure MAIN
    prompt for first name
    read first name
    add null terminator to first name
    if first name == 'Kevin' then
        print hello Kevin message
    else
        print disappointed message
    prompt for last name
    read last name
    add null terminator to last name
    add first name to full name
    add space to full name
    add last name to full name
    print full name
    print length of full name
```

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

int main(int argc, char **argv)
{
    // Declare character arrays to store name
    char first_name[50];
    char last_name[50];
    char full_name[50];

    // Prompt for first name
    printf("Please enter your first name: ");
    // Read name into array from stdin (standard input)
    fgets(first_name, 50, stdin);

    // Remove the newline character
    // Gets the length of the string.  Finds the null terminator
    int len = strlen(first_name);
    // Check if last character is a newline
    if (len > 0 && first_name[len - 1] == '\n')
        // If so set newline character to null terminator
        first_name[len - 1] = '\0';

    // Check if your name is Kevin
    if (strcmp(first_name, "Kevin") == 0)
        printf("Hey! Another Kevin\n");
    else
        printf("Oh well\n");

    // ***** DO THE SAME FOR LAST NAME *****

    // Join the strings
    // strcat looks for null terminator.  Set first character of full name
    full_name[0] = '\0';
    // Add first name to full name
    strcat(full_name, first_name);
    // Add a space
    strcat(full_name, " ");
    // Add last name
    strcat(full_name, last_name);
    // Print name
    printf("Your full name is %s which is %d characters long\n",
       full_name, strlen(full_name));

    return 0;
}
```

Listing 1.9: Reading from the Console

String Functions in C

We have used some of the first C functions from the `string.h` library. These have allowed us to work with the blocks of `char` memory data in a simple manner. There are more functions than just these, and you should do some further research if you are interested. The ones we have used are:

`strlen` - gets the length of a string

`strcmp` - compares two strings to see if they are equal. Returns 0 if the strings match. Returns < 0 if the first string comes before the second alphabetically. Returns > 0 if the first string comes after the second alphabetically. The *case* of the strings matters here. If you remember your ASCII, upper case letters have a higher value than lower case

`strcat` - adds the second string to the end of the first string. Addition is done at the first null terminator in the first string. All characters in the second string are added.

As we are dealing with raw memory, some of these functions can overwrite parts of memory you do not intend to. *Be careful of your data sizes!*.

1.4.1 Exercise

The code for this application is not complete, and you need to add the functionality to read in the last name (the part that says `// ***** DO THE SAME FOR LAST NAME *****`). It is almost identical to the first name reading code, except we are using the `last_name` variable. Once working and running, you should get an output similar to the following:

```
Please enter your first name: Kevin
Hey! Another Kevin
You entered 5 characters
Please enter your last name: Chalmers
You entered 9 characters
Your full name is Kevin Chalmers which is 14 characters long
```

Listing 1.10: Sample Output from Read Name Application

Variable Naming

We have declared our first variable data values in C!!!. As you can see it isn't hard. However, what you should be taking note of is the variable names used. *Variable names should describe the variable!.* Again, this falls under code quality, and it is important to use sensible consistent names throughout your code. There are a number of variable naming 'standards' and if you have one and are *consistent* then go ahead and use it. Just ensure your code is tidy and understandable.

What Happens When I Enter too Many Characters?

OK it is important to realise what `fgets` does when working with `stdin`. Lets say you enter 51 characters for your first name. What happens to the 51st character? Well it gets left on the buffer and is captured the next time you call `fgets` on `stdin`. *Try this out and see the result.* You need to understand these little issues to avoid common bugs in programming input-output based systems.

1.5 Conditionals and Loops

So far our applications have been very basic. We perform a number of operations in order then exit. We did introduce an `if` statement in the last piece of code, and didn't really explain it. You should have come across `if` statements by now, but just as a refresher.

What is an `if` Statement?

`if` is our most basic form of branching statement in C. It allows us to test a condition and execute a piece of code based on the value of that condition. If the condition value evaluates to true, then we execute the code within the brackets of the `if` statement.

```
if (condition)
{
    // If condition is true this code is run
    ...
}
```

If the condition evaluates to false, then we don't execute the code. When we want different code to run when the condition isn't true, we use the `else` statement:

```
if (condition)
{
    // If condition is true this code is run
    ...
}
else
{
    // If condition is false this code is run
    ...
}
```

We can combine multiple `if` statements together if we wish:

```
if (condition1)
{
    // If condition1 is true this code is run
    ...
}
else if (condition2)
{
    // If condition1 is false and condition2 is true this code
       is run
    ...
}
else
{
    // If condition1 is false and condition2 is false this code
       is run
    ...
}
```

`if` statements and branching / conditional operations are what we call `selection` operations. So far we have only been using *sequence* of operations. Adding *selection* allows us to run different code based on conditions. *Sequence* and *selection* provide two of the three main constructs of programming:

sequence - commands are executed in sequence.

selection - a choice (selection) of commands can be taken based on some conditional value. This allows different sequences of code to be run based on conditions.

iteration - a sequence of commands can be run more than once. This is supported by the use of *looping* statements.

Loops in C/C++

C and C++ support the same basic loops that Java supports. Just as a reminder these are:

`while(condition)` - the commands in the `while` block are executed as long as the given condition is true. This means that the loop may never run if `condition` is false

`do ... while(condition)` - the commands in the `do while` loop are executed as long as the given condition is true. The loop will execute at least once.

`for(value; condition; command)` - `for` loops allow us to perform some form of initialisation (the `value` part) at the start of the loop, test for a condition at the start of each loop iteration (the `condition` part), and perform a command at the end of each iteration (the `command` part).

We will come across examples of these as we work through the rest of the module.

Now we have introduced the basic ideas of looping and conditionals, let us build a simple application using these techniques. The application we are going to build

Algorithm 2 Star Printing Algorithm

```
procedure MAIN
    $flag \leftarrow 1$
    while $flag = 1$ do
        prompt for number of stars ($number$)
        read in number of stars ($number$)
        if $number = 0$ then
            $flag \leftarrow 0$
            continue
        for $i \leftarrow 0$ to $number$ do
            for $j \leftarrow 0$ to $i$ do
                print *
            print new line
    print goodbye
```

will print out a triangle of stars based on a number input. Algorithm 2 provides an overview of the program flow.

The code listing for this algorithm is below:

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

int main(int argc, char **argv)
{
    // Flag (1 or 0) to indicate if we should continue
    int flag = 1;
    // while loop
    while (flag)
    {
        // Character array to read number into
        char buffer[10];
        // Prompt for value
        printf("How many stars? (0 to quit) ");
        // Read in value
        fgets(buffer, 10, stdin);
        // Convert string to number
        int number = atoi(buffer);
        // If number equals 0 then set flag and continue
        if (!number)
        {
            flag = 0;
            continue;
        }
        // Print stars
        for (int i = 0; i < number; ++i)
        {
            for (int j = 0; j < i; ++j)
                printf("*");
            printf("\n");
        }
    }
    // Say goodbye
    printf("Goodbye!\n");

    return 0;
38 }
```

Listing 1.11: Printing Stars

True and False in C

Line 9 might seem strange to those who haven't used C before. We are not testing to see if the `flag` value is equal to 1. We are treating it like it is either a true or false value. This is one of problems that can effect new C programmers.

C does not have a `boolean` type as Java. It is actually quite a limitation. As such, C can treat any value as a `boolean` value. The following rules are used to determine if a value is true or false:

- Value equals 0 means that value is false
- Any other value means true – including negative values

To be fair, a `bool` type is provided by the `stdbool.h` header file (in the C99 standard and later), but loops and conditionals don't require you to use them - they use the above rules.

Treating boolean type values in this manner has it's advantages and disadvantages:

- An object that is `null` (we will look at this in more detail later in the module) then it is 0 (false). This allows us to write statements based on whether an object is valid or not. This can make code neater, but can be confusing to non-C/C++ programmers.
- The execution of an assignment statement evaluates to what was assigned into the left-hand side. *This can lead to many bugs in C/C++ programs* (although a good compiler should spot it).

On the last point, the following code is an example of one of the issues:

```
1 int x = 10;
2 if (x = 5)
3 {
4     // Note the single equals.  We assigned 5 to
5     // x, so the expression evaluates to 5, i.e. true.  The code
          in the if block
6     // will run!
7 }
```

The `if` condition on line 2 does equal true (1). We can assign 5 to `x`.

`break` and `continue`

Line 23 contains another keyword - `continue`. This keyword is used to control how a loop operates. There is another similar keyword - `break`.

`continue` - ends this iteration of the loop and allows the condition to be checked to see if another iteration should be executed.

`break` - exits the loop completely

The break statement is also used in case statements which we will look at shortly. These statements are useful to remember when you want to change the behaviour of a loop.

Converting a String to a Value

We have introduced a lot of new ideas in this piece of code! On line 18 we used another new function - atoi. This function converts a string to an integer (aSCII to integer). Why do we need to do this? Well there is a difference between a value and it's string representation. *This is another fundamental concept you need to understand!*. If you remember your ASCII codes (or you can look up an ASCII table), the character for '1' is equal to 49. This is how the computer stores the textual representation. We need to convert this to the actual value 1. The atoi function does this for us.

We will look at other conversion functions shortly.

Running this application provides output similar to the following:

```
How many stars? (0 to quit) 8

*
**
***
****
*****
******
*******
How many stars? (0 to quit) 4

*
**
***
How many stars? (0 to quit) 1

How many stars? (0 to quit) 0
Goodbye!
```

Listing 1.12: Output from Stars Application

It isn't quite working the way we want it to yet! Work through the following exercises to fix the problems.

1.5.1 Exercises

1. Fix the application so the correct number of stars are output. You will notice that we do not print a line of stars with the number entered. You will have to modify the conditional test on the for loop to fix this.

2. Modify the application so it doesn't need the flag value. To do this you will need to use the break statement.

1.6 case Statements - Creating a Menu

When we discussed if statements we introduced the idea of using multiple else if statements to combine different conditions. Although this is good for many general cases, we sometimes want to branch based on the value of a variable. This is where case statements come in useful. In particular, menus are a perfect example of using a case statement effectively.

As an example, let us build a simple menu application. Algorithm 3 outlines the basic behaviour.

Algorithm 3 Menu System

```
procedure MAIN
    $flag \leftarrow 1$
    while $flag = 1$ do
        print menu
        read input ($number$)
        switch $number$ do
            case 1
                print hello
            case 2
                print goodbye
            case 0
                print exit
                $flag \leftarrow 0$
            case default
                print error
```

The general idea is that we print a menu, get the input, and act on the input based on the menu options. The C code implementation of Algorithm 3 is given below:

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

int main(int argc, char **argv)
{
    int flag = 1;
    while (flag)
    {
        // Print menu
        printf("1 - say hello\n");
        printf("2 - say goodbye\n");
        printf("0 - exit\n");
        printf("Enter choice - ");
        // Read input
        char buffer[10];
        fgets(buffer, 10, stdin);
        // Convert to number
        int number = atoi(buffer);

        // Work on input
        switch (number)
        {
            case 1:
                printf("Hello World!\n");
                break;
            case 2:
                printf("Goodbye World!\n");
                break;
            case 0:
                printf("Exiting...\n");
                flag = 0;
                break;
            default:
                printf("*** INVALID INPUT ***\n");
                break;
        }
    }

    return 0;
}
```

Listing 1.13: Using `switch` and `case` Statements

> **The `default` Keyword**
>
> The `default` keyword is only called if none of the other cases are met. It is useful to deal with erroneous behaviour. It can be left out if you wish.

An example output from this application is given below:

```
1 - say hello
2 - say goodbye
0 - exit
Enter choice - 1
Hello World!
1 - say hello
2 - say goodbye
0 - exit
Enter choice - 2
Goodbye World!
1 - say hello
2 - say goodbye
0 - exit
Enter choice - 4
*** INVALID INPUT ***
1 - say hello
2 - say goodbye
0 - exit
Enter choice - 0
Exiting...
```

Listing 1.14: Output from Menu Application

1.6.1 Exercise

What will happen if you remove the `break` statements from the `switch-case` statements? Try this out. Do you understand the output?

1.7 Functions

We have looked at how we can use conditionals and loops to avoid writing lots of code, but what other techniques do we have? Hopefully you remember the idea of writing functions (or methods, procedures, sub-routines depending on how you were introduced to the idea). Functions allow us to separate our code into different, *reusable* sections. This is the foundation of working in a `structured programming` manner.

What is a Function?

We are going to use the more general definition of function in a programming context rather than the mathematical or functional programming idea. A function is just a collection of statements that we can run that returns some form of value (possibly `void` which means no value) given a set of parameters. In C, a function takes the following form:

`return-type` `function-name(`*`params`*`)`

The parts of the function definition are:

`function-name` - the name we use to call the function. For example, we have been using `printf` throughout our code.

`return-type` - the type of value returned by the function. We will look in more details at data types in the next unit.

`params` - a collection of parameters (with types) we pass to the function.

The point of a function (or procedure, method, sub-routine) is to allow us to call another piece of functionality. Usually we write functions that are reusable (have a particular purpose) and small (allow us to separate our code into easy to understand chunks). *Writing functions comes with practice!*. You might find it difficult to determine how to separate your code initially, but it will come with experience.

Parameter Passing

We will be covering parameter passing in far greater detail towards the middle of the module. Parameter passing is another area where novice programmers can slip up. At the moment you should consider that a parameter you pass into a function is a *copy*. Let us reiterate that idea - *parameters passed into a function are a copy of that value!*. This means that if you change the value in a function, this change is not reflected outside the function. The function has only changed its copy. There is a good chance you have worked with the idea of passing values by reference in Java and not thought about it. We will spend an entire unit of this module looking at parameter passing.

Let us build an example for working with functions. Our application will prompt a user for their name and then calculate their wage based on hours worked and hourly rate.

We will also write a short function (or procedure - it doesn't return a value) that will remove the newline character from our entered name. We did this before in our name application. We will also enable the modification of this value by passing it in as a *pointer*. Do not worry about this at the moment. We will introduce these ideas more and more through the units we are working through

```c
#include <stdio.h>
// The main C standard library header
#include <stdlib.h>
#include <string.h>

// Calculate wage
double calculate_wage(double rate, double hours)
```

Algorithm 4 Wage Calculator

```
function CALCULATE_WAGE($rate$, $hours$)
    return $rate \times hours$

procedure MAIN
    $flag \leftarrow 1$
    while $flag = 1$ do
        read name ($name$)
        if $name =$ ”0” then
            $flag \leftarrow 0$
            continue
        read rate ($rate$)
        read hours ($hours$)
        $wage \leftarrow$ CALCULATE_WAGE($rate$, $hours$)
        print wage
```

```c
{
  return rate * hours;
}

// Remove newline.  Pass in modifiable value
void remove_newline(char *str)
{
  // Gets the length of the string
  int len = strlen(str);
  // Check if last character is a newline
  if (len > 0 && str[len - 1] == '\n')
    // If so set newline character to null terminator
    str[len - 1] = '\0';
}

int main(int argc, char **argv)
{
  // Flag to continue
  int flag = 1;
  while (flag != 0)
  {
    // Buffer for name
    char name[50];
    // Buffer for number
    char number[10];
    // Prompt for first name
    printf("Please enter your name: ");
    // Read name into array
    fgets(name, 50, stdin);
    // Strip newline
    remove_newline(name);
    // If name is 0 then exit
    if (strcmp(name, "0") == 0)
    {
      flag = 0;
      continue;
    }
    // Prompt for hourly rate
    printf("Enter hourly rate: ");
    // Read value into array
    fgets(number, 10, stdin);
    // Convert to float
    double rate = atof(number);
    // Prompt for hours
    printf("Enter hours: ");
    // Read value into array
    fgets(number, 10, stdin);
    // Convert to float
    double hours = atof(number);
    // Calculate wage
    double wage = calculate_wage(rate, hours);
    // Output
    printf("Wage for %s: %.2f\n", name, wage);
  }
  // Print goodbye
  printf("Goodbye!");

  return 0;
}
```

Listing 1.15: Creating and Calling a Function

The `stdlib.h` Header File

We have added a new header file - `stdlib.h`. This is the STanDard LIBrary header file, and contains quite a collection of useful functions - this is actually where the string conversion functions reside. However, it should be one of the headers you always include in a C application. In general, the following headers are the ones you should generally include as default:

- `stdlib.h`
- `stdio.h`
- `string.h` - if you are using text at all

The `atof` Function

We already used the `atoi` function. `atof` converts a string to a floating point (decimal) number (aSCII to float). `atof` returns a `double` value (double precision floating point). We will look at data types in the next unit.

Declaration Order

C and C++ are not as forgiving as Java and other languages when it comes to declaration order. Before you can use a function or variable it has to be declared. This is a question of scope. We will be covering scope in more detail later in the module. At the moment realise that if you want to use a function or variable at a point in your file, *it must be declared earlier in the file than where you use it.* Otherwise you will get an error. The exercise will illustrate this.

An example output from this application is shown below.

```
Please enter your name: Kevin
```

```
Enter hourly rate: 35
Enter hours: 35
Wage for Kevin: 1225.00
Please enter your name: Bob
Enter hourly rate: 50
Enter hours: 40
Wage for Bob: 2000.00
Please enter your name: Frank
Enter hourly rate: 100
Enter hours: 10
Wage for Frank: 1000.00
Please enter your name: 0
Goodbye!
```

Listing 1.16: Output from Wage Calculation Application

1.7.1 Exercise

Move the `calculate_wage` function after `main` and compile the program to see what happens. You should get an

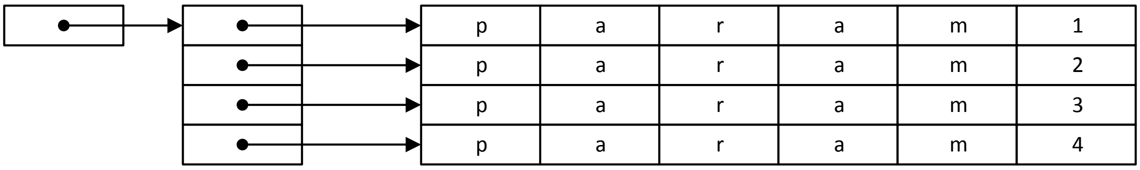
```
Microsoft (R) Program Maintenance Utility Version 12.00.21005.1
Copyright (C) Microsoft Corporation.  All rights reserved.

        cl functions.c
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.30501 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

functions.c
functions.c(64) : error C2371: 'calculate_wage' : redefinition; different basic
types
NMAKE : fatal error U1077: '"C:\Program Files (x86)\Microsoft Visual Studio 12.0
\VC\BIN\cl.EXE"' : return code '0x2'
Stop.
```

1.8 Command Line Arguments

For our final application before moving onto what the compiler is doing let us examine the two values that we declare for our main application - `argc` and `argv`.

`argc` provides a count of the number of command line arguments that have been passed into the application. It is an integer value that provides information about its partner value - `argv`.

`argv` is an interesting parameter. Its type is `char**`, or a pointer to a pointer of `char`. If you remember from our work on strings at the start of the unit, we defined a string as a `char*`. We also defined an array of `char` as a `char*`. A `char**` can be considered as an `array of strings`. `argc` tells us how many strings we have. As the strings are null terminated, we can use `strlen` to get the length of each string if we want. Figure 1.4 illustrates the general idea.



p	a	r	a	m	1
p	a	r	a	m	2
p	a	r	a	m	3
p	a	r	a	m	4

Figure 1.4: Command Line Arguments Structure

As an example of working with the command line arguments, try the following (fairly short) application.

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

int main(int argc, char **argv)
{
  // Loop for number of arguments
  for (int i = 0; i < argc; ++i)
  {
    printf("Argument %d: %s\n", i, argv[i]);
  }
  printf("All arguments printed.\n");

  return 0;
}
```

Listing 1.17: Outputting Command Line Arguments

Running this application (let us assume you have called it `command-line`) provides the following output:

```
command-line hello world programming fundamentals
Argument 0: command-line
Argument 1: hello
Argument 2: world
Argument 3: programming
Argument 4: fundamentals
All arguments printed.
```

Listing 1.18: Printing Command Lines

Notice that the name of the application is argument 0. This does depend on the operating system running the application. Windows provides the name as an argument. This is worth remembering.

1.9 Compiling a Source File

We have done a quick overview of a number of basic programming concepts you should have been familiar with prior to the module (and possibly some new concepts). Now to look at why this unit is called *Compiling and Linking Programs*. So far we have been doing both of these stages at once. Now we will look at how these stages work together.

When we program, we create some form of textual file that contains our commands written in some form of high level language (normally). C is the high level language we are working in at the moment. We then use some form of compiler to transform our code into something that our computer can understand. This is the compilation step.

When it comes to native code (such as written in C and C++) there is actually two stages:

1. Compile the code into an *object* (`.obj`) file. This is an intermediate file that has machine code within it but cannot be run - it does not know how

2. Link a collection of *object* files together to create an executable. The linking process will also combine other libraries which allows the code to communicate with the operating system.

The second stage is quite important to allow our application to run, and it adds a lot of extra code. To illustrate this, look at the size of our Hello World application and the parts built:

- `hello.c` is 378 bytes in size
- `hello.obj` is 651 bytes in size
- `hello.exe` is 73216 bytes in size!

`hello.exe` is over 100 times larger that the initial file. There is a lot of extra code added in the final application. This is the linking step.

Essentially we have the following steps occurring:

C code ⇒ object file ⇒ executable

Let us go through these two steps. We will work using the `hello.c` code we created at the start of the unit. To compile our code (produce an object file only) we use the following command:

```
cl /c hello.c
```

Listing 1.19: Compiling to a Object File

This will provide the following output:

```
cl /c hello.c
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.30501 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

hello.c
```

Listing 1.20: Output from Object File Compilation

It is as simple as that. You can take a look at the `.obj` file but it will just look like numbers. A Hex code viewer will show you a bit more.

1.10 Linking an Object File

So we have an object file, how do we create an executable file? For this we use the `link` command. To create our Hello World executable from our `hello.obj` we use the following command:

```
link hello.obj
```

Listing 1.21: Linking an Object File

The output from this command is shown below:

```
link hello.obj
Microsoft (R) Incremental Linker Version 12.00.30501.0
Copyright (C) Microsoft Corporation.  All rights reserved.
```

Listing 1.22: Output from Linking Stage

Not much to see here, but we have linked our object file and created an executable. We will look at the linking step in more detail later in the module when we build libraries.

1.11 Outputting Assembly Code

In a way, the building of an object file is also a two step process. C is considered a good systems programming language as it has a very close resemblance to assembly code. We can make the `cl` compiler output assembly code for us to see the result.

What is Assembly Code?

You will have hopefully been exposed to assembly code in Computer Systems. Assembly code is a language which provides human readable versions of machine level instructions. We will be looking in more detail at how the CPU executes instructions and writing some assembly code later in the module. What you will find is that you can translate from C to assembly fairly easily.

We can extend our compile pipeline to the following:

C code ⇒ Assembly code ⇒ Object code ⇒ executable

Let us go through this extra step using our Hello World example. Just as a refresh, the code for this is given below:

```c
// stdio contains standard C input-output functions
#include <stdio.h>

// Our main function.
// argc is the number of command line arguments
// argv are the command line arguments
// We will look at these later
int main(int argc, char **argv)
{
    // Print hello world to the screen
    printf("Hello world!\n");
    // Return value from program
    return 0;
}
```

To output assembly code during our compilation step we use the `/Fa` compiler flag:

```
cl /c /Fa hello.c
```

Listing 1.23: Generating Assembly Code

The output from this step is as follows:

```
cl /c /Fa hello.c
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.30501 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

hello.c
```

Listing 1.24: Output from Assembly Generation

Not really much difference for output than during our standard compile. However, you will now find that you have a `hello.asm` file produced. It should look similar to the following:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 18.00.30501.0

  TITLE D:\programming-fundamentals\code\unit-01\hello.c
  .686P
  .XMM
  include listing.inc
  .model  flat

INCLUDELIB LIBCMT
```

```
10 INCLUDELIB OLDNAMES
11
12 _DATA SEGMENT
13 $SG3048 DB  'Hello world!', 0aH, 00H
14 _DATA ENDS
15 PUBLIC  _main
16 EXTRN _printf:PROC
17 ; Function compile flags: /Odtp
18 _TEXT SEGMENT
19 _argc$ = 8             ; size = 4
20 _argv$ = 12            ; size = 4
21 _main PROC
22 ; File d:\programming-fundamentals\code\unit-01\hello.c
23 ; Line 9
24   push  ebp
25   mov ebp, esp
26 ; Line 11
27   push  OFFSET $SG3048
28   call  _printf
29   add esp, 4
30 ; Line 13
31   xor eax, eax
32 ; Line 14
33   pop ebp
34   ret 0
35 _main ENDP
36 _TEXT ENDS
37 END
```

Listing 1.25: Assembly Code for Hello World Application

OK, this might seem daunting, but we can actually break this down a little.

- Line 13 stores the Hello World message and allocates it an identity (the $SG3048 part). It has the characters of the message, followed by the hex values 0a (10) and 00 (as you might guess 0). The h after these values denote them as hex. If you look up an ASCII table, you will see that 10 is a line feed (the new line character) and 0 is null (the null terminator). We discussed these ideas back in the strings example.

- Line 21 is the start of the main procedure.

- Line 27 and 28 shows how the call to printf is made. The value stored in $SG3048 is pushed onto the stack and printf is called. The stack is where we store arguments to pass to functions. We will revisit the stack and scope later in the module

The lines starting with ; are comments. Some of these comments tell you where these lines of code relate to our original C.

To compile our assembly code we use the ml program:

```
ml /c <filename.asm>
```

Listing 1.26: Compiling Assembly Code

If we compile our hello.asm code we get the following output:

```
ml /c hello.asm
Microsoft (R) Macro Assembler Version 12.00.30501.0
Copyright (C) Microsoft Corporation.  All rights reserved.
```

```
Assembling: hello.asm
```

Listing 1.27: Output from Assembly Code Compilation

You can now link the produced `hello.obj` as before.

1.12 Exercises

These exercises are meant to reaffirm your understanding the concepts covered in this unit, and you will find that you will become a much better program via practice and by solving these problems.

1. Write an application that outputs your name surrounded by stars. Algorithm 5 provides the outline. An example output is:

```
******************
* Kevin Chalmers *
******************
```

2. Write an application that prompts for a student name and then asks for a grades. The application for keep asking for grades until -1 is entered. Once -1 is entered, an average grade should be given. The application should ask for student names until 0 is entered as a student name.

3. Write an application that outputs a Christmas tree of a given height. For example, given 5 as an input:

```
    *
   ***
  *****
 *******
*********
    *
```

4. Write an application that presents a menu asking if the user wants a Christmas tree or a triangle. The application will then ask for a size and print the requested object. The application should keep asking the user what they want to print until asked to exit. The Christmas tree and triangle code should be written as functions.

5. Generate assembly for some of the other applications and examine the output. Try and compare what is going on in the assembly code and the C code. We will be looking at this in more detail in two units time.

Algorithm 5 Printing Your Name Surrounded by Stars

```
procedure MAIN
    read name ($name$)
    $length \leftarrow \textsc{strlen}(name) + 4$
    for $i \leftarrow 0$ to $length$ do
        print $*$
    print new line
    print $*$
    print space
    print name
    print space
    print $*$
    print new line
    for $i \leftarrow 0$ to $length$ do
        print $*$
```

Unit 2

Data Sizes and Data Representation

We mentioned in the last unit that C is often referred to as a systems language. This is because it provides a close approximation to how the computer works. As such, when we declare data in C it has a direct representation in memory. This has advantages when working with memory, but does mean we need to be more cautious in comparison to languages such as Java and C# (which are often referred to as *managed languages*).

When it comes to working with data in C, the first idea we have to understand is how data is represented and how to find out the sizes of our data types. Memory size is also an important consideration when worrying about system performance and limitations.

2.1 Getting the Size Of Data Using `sizeof`

How big is a value? Well, that does actually depend on the system you are working in and the compiler you are using. In C and C++ we can use the `sizeof` function to get the size of a data type in bytes.

The `sizeof` Function

`sizeof` is a very useful function we will use when we work with memory allocation in the second half of the module. It requires either a type or a value and will return the number of bytes used to represent the type or value.

`sizeof` returns a value of type `size_t`. Although this type is compiler specific, it is normally of type `unsigned int`. We will typically just treat it as `int`.

Let us now test the size of the standard `int` type. The following application will print this value out for us.

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

int main(int argc, char **argv)
{
    printf("The size of int is %d bytes\n", sizeof(int));

    return 0;
}
```

Listing 2.1: Using `sizeof`

Running this application will provide an output similar to the following.

```
The size of int is 4 bytes
```

Listing 2.2: Output from `sizeof` Application

Depending on the system you are working in you might get a different result. However, it is pretty standard now that an `int` value is 4 bytes. We will look at what this actually means for maximum and minimum values shortly.

2.2 Unsigned Data Types

One of the key differences between C and Java is that we can define values as being either `signed` or `unsigned`. A `signed` value is the default type. These means a value can represent both positive and negative values. An `unsigned value` means that only positive numbers can be represented.

`signed` versus `unsigned` Value Representation

We aren't going to cover this in any great detail, but you should have covered the concepts of *twos* complement and signed bits. Let us consider a simple 8-bit value. In a signed value, the first bit is the signed bit. If the first bit is 0, the value is positive:

$$00000000$$

If the first bit is 1, the value is negative:

$$10000000$$

The other 7 bits represent the value of the 8-bit number. This means that we have two values for 0 - positive 0 and negative 0 - as illustrated in our two examples.

The following application will give you an idea of what happens when we work with `unsigned` values.

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

int main(int argc, char **argv)
{
    int a = -500;
    unsigned int b = -500;
    unsigned int c = (unsigned int)a;
    float f = -500;

    printf("a signed: %d, a unsigned: %u\n", a, a);
    printf("b signed: %d, b unsigned: %u\n", b, b);
    printf("c signed: %d, c unsigned: %u\n", c, c);
    printf("f signed: %d, f unsigned: %u\n", f, f);
    printf("a == b: %d\n", a == b);
    printf("a == c: %d\n", a == c);
17     printf("a == f: %d\n", a == f);
18 
19     return 0;
20 }
```

Listing 2.3: Unsigned Data Types

Printing `unsigned` Values

To print a value as `unsigned` with `printf` we use the `%u` placeholder. This can be combined with other data types to print them as `unsigned`. We will look at this again shortly.

On line 7 and 8 we declare our `unsigned values` - on line 7 the value is even set as negative. Our `printf` statement tries to print all the values as `signed` and `unsigned`. We also print out if the values are equal. Running this application provides the following output:

```
a signed: -500, a unsigned: 4294966796
b signed: -500, b unsigned: 4294966796
c signed: -500, c unsigned: 4294966796
f signed: 0, f unsigned: 3229564928
a == b: 1
a == c: 1
a == f: 1
```

Listing 2.4: Output from Unsigned Application

Remember that C represents false as 0 and true as any other value - in this case 1.

Notice that `printf` treats all the `int` values as if they were the same value. This is because they have the same bit representation. C has undertaken a rather crude conversion between `signed` and `unsigned` values. The only difference is when we print the `float` value. `float` values have a different bit representation and therefore when we attempt to print it as a `signed` or `unsigned int` then we get a different result.

Looking at the equality tests though shows us that C does do more work here. `a` is shown to be equal to `f`, although the bit representations are different. This is because C will attempt to convert the values to the same type. There are rules that govern this conversion. You will learn them through practice.

2.3 Minimum and Maximum Values of Data Types

Given that we can work out the size of a data type in bytes using `sizeof` and know the difference between `signed` and `unsigned` values and the overhead of having `signed` values, we are able to work out the maximum and minimum values of each type. We could do this by hand, but we would have to do it for every system we compiled our programs for. This isn't very efficient either. Thankfully we have the `limits.h` header file which provides us.

`limits.h`

The `limits.h` header file contains a collection of different values that define the minimum and maximum values of the standard types. We will be looking at these shortly. If you need to know the maximum or minimum of the standard values then include the `limits.h` header.

Let us do a quick test of finding out the limits. Run the following application:

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

int main(int argc, char **argv)
{
    printf("unsigned int is %d bytes, min value %u, max value %u\n"
        , sizeof(unsigned int), 0, UINT_MAX);
    printf("int is %d bytes, min value %d, max value %d\n", sizeof(
        int), INT_MIN, INT_MAX);

    return 0;
}
```

Listing 2.5: Getting Minimum and Maximum Values

Notice that for `unsigned int` we use 0 as the minimum - there is no other minimum defined. The other values (`UINT_MAX`, `INT_MIN` and `INT_MAX`) are all defined in the `limits.h` header file. Running this application will give the following result (based on your system).

```
unsigned int is 4 bytes, min value 0, max value 4294967295
int is 4 bytes, min value -2147483648, max value 2147483647
```

Listing 2.6: Output from Min-Max Application

So we now know how to get the size of a value and also how to get the minimum and maximum values. Let us combine these ideas into a single application.

2.4 Different Datatypes

The following application will print the sizes, minimum, and maximum values of the standard C types. We will use `float.h` to get the minimum and maximum values of the floating point number types as well. The application is below. It also includes pointer values.

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

int main(int argc, char **argv)
{
    printf("unsigned char is %d bytes, min value %u, max value %u\n
        ", sizeof(unsigned char), 0, UCHAR_MAX);
    printf("signed char is %d bytes, min value %d, max value %d\n",
         sizeof(signed char), SCHAR_MIN, SCHAR_MAX);
    printf("char is %d bytes, min value %d, max value %d\n", sizeof
        (char), CHAR_MIN, CHAR_MAX);
    printf("unsigned short is %d bytes, min value %u, max value %d\
        n", sizeof(unsigned short), 0, USHRT_MAX);
    printf("short is %d bytes, min value %d, max value %d\n",
        sizeof(short), SHRT_MIN, SHRT_MAX);
    printf("unsigned int is %d bytes, min value %u, max value %u\n", sizeof(unsigned int), 0, UINT_MAX);
    printf("int is %d bytes, min value %d, max value %d\n", sizeof(int), INT_MIN, INT_MAX);
    printf("unsigned long is %d bytes, min value %lu, max value %lu\n", sizeof(unsigned long), 0, ULONG_MAX);
    printf("long is %d bytes, min value %ld, max value %ld\n", sizeof(long), LONG_MIN, LONG_MAX);
    printf("unsigned long long is %d bytes, min value %llu, max value %llu\n", sizeof(unsigned long long), 0ULL, ULLONG_MAX);
    printf("long long is %d bytes, min value %lld, max value %lld\n", sizeof(long long), LLONG_MIN, LLONG_MAX);
    printf("float is %d bytes, min value %e, max value %e\n", sizeof(float), FLT_MIN, FLT_MAX);
    printf("double is %d bytes, min value %e, max value %e\n", sizeof(double), DBL_MIN, DBL_MAX);
    printf("long double is %d bytes, min value %e, max value %e\n", sizeof(long double), LDBL_MIN, LDBL_MAX);

    // Pointer sizes
    printf("char* is %d bytes\n", sizeof(char*));
    printf("short* is %d bytes\n", sizeof(short*));
    printf("int* is %d bytes\n", sizeof(int*));

    return 0;
}
```

Listing 2.7: Displaying Information from Different Datatypes

More `printf` Placeholders

We have introduced two new placeholders for `printf` this time. These are as follows:

`%l` - placed before another placeholder means that we expect a `long` version of the value (`signed` or `unsigned`). Can be used with another long placeholder for `long long` values.

`%e` - prints out the value in scientific notation format

Why no `unsigned` Floating Points?

We have used the floating point values in this application, but we haven't defined any `unsigned` floating point values. This is because they don't exist. This is because the CPU does not support `unsigned` floating point values, therefore neither does C. All floating point values are `signed`.

Running this application provides the following output:

```
unsigned char is 1 bytes, min value 0, max value 255
signed char is 1 bytes, min value -128, max value 127
char is 1 bytes, min value -128, max value 127
unsigned short is 2 bytes, min value 0, max value 65535
short is 2 bytes, min value -32768, max value 32767
unsigned int is 4 bytes, min value 0, max value 4294967295
int is 4 bytes, min value -2147483648, max value 2147483647
unsigned long is 4 bytes, min value 0, max value 4294967295
```

```
long is 4 bytes, min value -2147483648, max value 2147483647
unsigned long long is 8 bytes, min value 0, max value 18446744073709551615
long long is 8 bytes, min value -9223372036854775808, max value 9223372036854775807
float is 4 bytes, min value 1.175494e-038, max value 3.402823e+038
double is 8 bytes, min value 2.225074e-308, max value 1.797693e+308
long double is 8 bytes, min value 2.225074e-308, max value 1.797693e+308
char* is 4 bytes
short* is 4 bytes
int* is 4 bytes
```

Listing 2.8: Output from Data Types Application

Pointers and Pointer Sizes

The pointer values are all 4 bytes in size. This is because we have been building 32-bit applications. A 32-bit application uses 32-bit (*4 byte*) values to represent memory locations, and a pointer is just a memory location. If you calculate the maximum number of addressable memory locations using the standard calculation (2^{32}) you end up with 4294967296 bytes - or 4 GB. You might recognise this as the maximum amount of memory that a 32-bit operating system supports. It is also the maximum amount of memory that a 32-bit application can allocate.

Table 2.1 provides an overview of the integer data types and their sizes based on our result of running this program using the Microsoft compiler on Windows. Note that these data sizes are not consistent across other operating systems and compilers. You should always check your data sizes in the environment you are working in.

Type	**Size (bytes)**	**Minimum Value**	**Maximum Value**
`unsigned char`	1	0	$2^8 - 1$
`signed char`	1	$-2^7 - 1$	$2^7 - 1$
`unsigned short`	2	0	$2^{16} - 1$
`signed short`	2	$-2^{15} - 1$	$2^{15} - 1$
`unsigned int`	4	0	$2^{32} - 1$
`signed int`	4	$-2^{31} - 1$	$2^{31} - 1$
`unsigned long`	4	0	$2^{32} - 1$
`signed long`	4	$-2^{31} - 1$	$2^{31} - 1$
`unsigned long long`	8	0	$2^{64} - 1$
`singed long long`	8	$-2^{63} - 1$	$2^{63} - 1$

Table 2.1: Data Sizes, and Minimum and Maximum Values of Integer Types

2.5 Compiling for 64-Bit

Let us rebuild this application in 64-bit to see what the difference is. To do this, you have to run a different command prompt. You will find it under *Visual Studio 2013*, *Visual Studio Tools*. The command prompt you want is *VS2013 x64 Native Tools Command Prompt*. From here, it is just the same process as before. Use `cl` to compile the applications. If you run the previous application compiled using the 64-bit compiler you will get the following output:

```
unsigned char is 1 bytes, min value 0, max value 255
signed char is 1 bytes, min value -128, max value 127
char is 1 bytes, min value -128, max value 127
unsigned short is 2 bytes, min value 0, max value 65535
short is 2 bytes, min value -32768, max value 32767
unsigned int is 4 bytes, min value 0, max value 4294967295
int is 4 bytes, min value -2147483648, max value 2147483647
unsigned long is 4 bytes, min value 0, max value 4294967295
long is 4 bytes, min value -2147483648, max value 2147483647
unsigned long long is 8 bytes, min value 0, max value 18446744073709551615
long long is 8 bytes, min value -9223372036854775808, max value 9223372036854775807
float is 4 bytes, min value 1.175494e-038, max value 3.402823e+038
double is 8 bytes, min value 2.225074e-308, max value 1.797693e+308
long double is 8 bytes, min value 2.225074e-308, max value 1.797693e+308
char* is 8 bytes
short* is 8 bytes
int* is 8 bytes
```

Listing 2.9: Output from Data Types Application Compiled for 64-bit

This time the only difference is the number of bytes allocated to a pointer. It is now *8 bytes*. This makes sense - we are using 64-bit code now. You can work out how much addressable memory this means you have now.

2.6 Strings

We previously discussed how C manages strings of characters in Section 1.3. Now let us look at how the different methods of creating a string affect the data sizes allocated. Remember we had a few different methods of declaring a string. We will look at two separate techniques:

1. Declaring a pointer to a `char` - a `char*`
2. Declaring a fixed size array of `char` - a `char[]`

C treats these two approaches in different ways. Let us write a test application to see what happens:

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

int main(int argc, char **argv)
{
    // Declare some strings
    char *hello = "Hello World!";
    char goodbye[15] = "Goodbye World!";

    // Print the messages
    printf("Hello message: %s\n", hello);
    printf("Goodbye message: %s\n", goodbye);

    // Get their size
    printf("Size of hello message: %d bytes\n", sizeof(hello));
    printf("Size of goodbye message: %d bytes\n", sizeof(goodbye));

    // Get the size of the data pointed to by the string
    printf("Size of data in hello message: %d bytes\n", sizeof(*
       hello));
    printf("Size of data in goodbye message: %d bytes\n", sizeof(*
       goodbye));
22
23     // Get the length of the string
24     printf("Length of hello message: %d bytes\n", strlen(hello));
25     printf("Length of goodbye message: %d bytes\n", strlen(goodbye));
26
27     return 0;
28 }
```

Listing 2.10: Data Sizes and Strings

Dereferencing a Pointer
On lines 20 and 21 we have used the following pieces of code:

- `*hello`

- `*goodbye`

When we use the star (*) in front of a pointer variable, we are asking to *dereference* the pointer. This means that we want to access the *data* stored in the pointed to memory location rather than just the pointer. We will be covering pointers later in the module, so you don't have to worry about this idea at the moment.

Running this application will give you the following output:

```
Hello message: Hello World!
Goodbye message: Goodbye World!
Size of hello message: 4 bytes
Size of goodbye message: 15 bytes
Size of data in hello message: 1 bytes
Size of data in goodbye message: 1 bytes
Length of hello message: 12 bytes
Length of goodbye message: 14 bytes
```

Listing 2.11: Output from String Sizes Application

Let us look at the 6 sizes output:

1. Hello message is 4 bytes as it is a pointer to a string of character data. Remember that a pointer is 4 bytes in size on a 32-bit machine (if you compile this in 64-bit the answer will be 8 bytes).

2. Goodbye message is 15 bytes long. This is because we have created a fixed size array, which C knows the size of.

3. Size of data in hello is 1 byte. We have dereferenced the pointer to access the data which is of type `char`. Remember that a `char` is 1 byte in size.

4. Size of data in goodbye is the same as hello.

5. Length of hello is 12 bytes. The message is 12 `char` long.

6. Length of goodbye is 14 bytes. The message is 14 `char` long

So the same general rules apply to sizing strings as they do to sizing other data types. The new interesting value is the array size. *This only works for fixed size arrays.* We will look at arrays in far more detail as we go through the module, and understand the limitations C and C++ has when working with array data.

2.7 Casting Between Types

So we have looked at data sizes and we understand how numbers are represented on the machine (to a basic level - *Computer Systems* teaches you more). So what about converting a number between different types and representations? Well to do this we use a technique called *casting*. Casting becomes quite a big concept when we work in object-orientation and we will cover some of these ideas later in the module. However, let us look at how we cast numbers.

Casting

Casting is the operation of converting one type of value to another. It actually happens all the time when you are programming. A lot of the time you just don't see it. We will discuss automatic casting shortly.

To perform a cast we use the following syntax:

type casted_value = (*type*)original_value;

The C compiler will take care of the necessary operations to convert from one type to another if necessary, although sometimes no specific operation occurs. As an example, if we wanted to convert form a `int` to a `float` we would do the following:

```c
int x = 10;
float y = (float)x;
// y is now 10.0f
```

However, if we performed the same operation trying to convert `x` to a `char*` you won't be converting an `int` to a string:

```c
int x = 10;
char *y = (char*)x;
// y now points to memory location 10!!!
```

In this instance, an `int` is converted to a memory location. Your pointer is now pointing at memory location 10. This is a piece of memory you cannot manipulate (it will be controlled by the operating system). If you try and print the data at this memory location your application will crash.

Casting is a useful operation when working with different number types. However, it does have its limitations as our test application will explore. In particular you need to think about what happens when you take a large number (say a `long long`) and convert it to a shorter number (say a `short`). What happens to the data that is lost? What does the number become? Our test application will examine this.

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

int main(int argc, char **argv)
{
    // Let's start with a unsigned char
    unsigned char c = UCHAR_MAX;
    printf("Value is %d\n", c);
    // Cast to signed char
    printf("As signed char %d\n", (char)c);
    // Cast to a short
    printf("As short %d\n", (short)c);
    // Cast to a int
    printf("As int %d\n", (int)c);
    // Cast to a long
    printf("As long %ld\n", (long)c);
    // Cast to long long
    printf("As long long %lld\n", (long long)c);

    // Now try a long long
    long long l = LLONG_MAX;
    printf("Value is %lld\n", l);
    // Cast to long
    printf("As long %lld\n", (long)l);
    // Cast to int
    printf("As int %lld\n", (int)l);
    // Cast to short
    printf("As short %lld\n", (short)l);
    // Cast to char
    printf("As char %lld\n", (char)l);

    // Now with a float
    float f = 555.5555f;
    printf("Value is %f\n", f);
    // Cast to char
    printf("As char %d\n", (char)f);
    // Cast to short
    printf("As short %d\n", (short)f);
    // Cast to int
    printf("As int %d\n", (int)f);

    return 0;
}
```

Listing 2.12: Casting Between Types

Running this application will give you the following output:

```
Value is 255
As signed char -1
As short 255
As int 255
As long 255
As long long 255
Value is 9223372036854775807
As long -1
As int -1
As short -1
As char -1
Value is 555.555481
As char 43
As short 555
As int 555
```

Listing 2.13: Output from Casting Application

For our `unsigned char` value, we first convert it to a `signed char`. This makes it -1! What has happened here? Well, a `signed char` has a maximum value of 127 with a signed bit. This is represented by the following binary string:

0111 1111

For a `unsigned char` the value 255 is represented by the following binary string:

1111 1111

When converting this second binary string to a `signed char`, using standard negative number conversion (remember your *Computer Systems* work) this gives as

-1. Every other data value is large enough to take the binary digits. Table 2.2 illustrates how the number is represented in binary for each of the different data types. It also shows how C converts the numbers (very trivially for `int` types).

Type	**Binary String**	**Value**
`unsigned char`	1111 1111	255
`signed char`	1111 1111	−127
`short`	0000 0000 1111 1111	255
`int`	0000 0000 0000 0000 0000 0000 1111 1111	255
`long`	0000 0000 0000 0000 0000 0000 1111 1111	255
`long long`	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111	255

Table 2.2: Converting `unsigned char` 255 to Other Types

When we go in reverse we are converting the `long long` value 9223372036854775807 (all binary digits 1). In this instance we just remove binary digits until we reach the desired size. As we are dealing with signed values this equates to -1 in each instance.

Type	**Binary String**	**Value**
`long long`	0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111	9223372036854775807
`long`	1111 1111 1111 1111 1111 1111 1111 1111	−1
`int`	1111 1111 1111 1111 1111 1111 1111 1111	−1
`short`	1111 1111 1111 1111	−1
`char`	1111 1111	−1

Table 2.3: Converting `long long` 9223372036854775807 to Other Types

For the `float` conversions more work is done since a `float` is not represented in this way in binary. Floating point conversion is therefore more likely to convert to a close approximation of the value required, although note that numbers are not rounded up but down. The conversion to `char` follows standard binary conversion which is why the value is 43. This is because C number conversion with floating point values follow certain rules:

- Find the smallest data type that the values will convert into with enough bits to get the entire value
- Convert value to common data type
- Convert to target data type

So, for example, the conversion from `float` to `char` is as follows:

$$float \Rightarrow int \Rightarrow char$$

Or in binary:

binary float ⇒ 0000 0000 0000 0000 0000 0010 0010 1011 ⇒ 0010 1011

Value conversion is an important idea to grasp, and in particular when we move onto object-oriented programming with C++. C does support more complex data types as well using the `struct` keyword. Let us look at that next.

2.8 Defining structs

Having simple data types only provides us with so much capability. In the real world we want to construct more complex data types to support our applications. We do this using the `struct` keyword. A `struct` is just a collection of values that we consider as a single value.

2.8.1 A struct in Memory

Let us consider a simple example that we will use in our test application. This `struct` will contain student details, and we define it as follows:

```
struct student
{
    unsigned int matric;
    char *name;
    char *address;
};
```

The student data type contains three values:

- A `unsigned int` used to represent the matriculation number of the student
- A `char*` (string) used to represent the student's name
- A `char*` (string) used to represent the student's address

We treat an variable of type `student` more or less as a normal data type, with some exceptions that we will cover shortly. First let us consider how a value of type `student` is stored in memory. Figure 2.1 illustrates how the bytes are laid out to store the `student struct`.

As can be seen, our data takes 12 bytes of storage, and is stored in the sequence of its declared member values. There is no additional overhead in C or C++ for storing data in a `struct`. We are just stating that a variable is of that type, and therefore the number of required bytes are stored.

2.8.2 Accessing struct Members

To access a value in a `struct` we use the dot notation (`.`) to allow us to access members of the `struct`. This is the same notation you are likely to remember from using Java. This is a simple method to allow us to use a variable name and the name of the member to access its value:

`variable.member_name`

We can get and set the member using this technique. We will explore these concepts far more later in the module.

Byte	Data
0	matric
1	matric
2	matric
3	matric
4	name
5	name
6	name
7	name
8	address
9	address
10	address
11	address

Figure 2.1: `student struct` Represented in Memory

2.8.3 Declaring a `struct` Variable

In C (not C++ as we will discover later in the module), we declare a variable of a `struct` type as follows:

`struct <type> <name>;`

It is much the same as a standard variable declaration, except that we have to place the keyword `struct` in front of the type. Otherwise, we have declared the variable as normal, and can access the values accordingly.

2.8.4 Passing `struct` Values as Parameters

As when declaring a `struct` variable we must also use the keyword `struct` when passing as a parameter. Again this is only in C. Basically a parameter pass is the same as declaring a variable for the function. It is just that we are providing that variable when we call the function.

2.8.5 `struct` Test Application

Our test application for working with `struct` data types is below. We are using our `student` data type, setting the values, and then printing the data type.

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

// A data structure containing student information
struct student
{
    unsigned int matric;
    char *name;
    char *address;
};

// Prints student information
void print_student(struct student s)
{
    printf("matric no: %u\n", s.matric);
    printf("name: %s\n", s.name);
    printf("address: %s\n", s.address);
}
```

```
19
20 int main(int argc, char **argv)
21 {
22     // Output size of the struct
23     printf("Size of student struct is %d bytes\n", sizeof(struct
           student));
24     // Create a student data structure
25     struct student s;
26     s.matric = 42001290;
27     s.name = "Kevin Chalmers";
28     s.address = "School of Computing";
29     // Print student data
30     print_student(s);
31
32     return 0;
33 }
```

Listing 2.14: A First `struct` Example

On line 23 we print the size of the `student struct`. Based on what we know about the value we should be able to work this out:

- the `matric` value is an `unsigned int` which is 4 bytes in size
- the `name` value is a `char*` which is 4 bytes in size
- the `address` value is a `char*` which is 4 bytes in size

Thus our `student` value should be 12 bytes in size. Running this application (below) shows that this is the case.

```
Size of student struct is 12 bytes
matric no: 42001290
name: Kevin Chalmers
address: School of Computing
```

Listing 2.15: Output from `struct` Application

Building complex data types that represent our real world data is very important. We will investigate this further when we look at object-orientation later in the module. However, we will use `struct` data occasionally from this point onwards.

2.9 `enum` Values and `case` Statements

Another data type we can declare is an `enum`. An `enum` allows us to declare a collection of options that a value could take. This is very useful for having easily read (to the programmer) code for ideas such as menu entries and system states. An `enum` can replace a numerical value for this case.

An example `enum` declaration is given below:

```
enum CHOICE
{
  EXIT = 0,
  HELLO = 1,
  GOODBYE = 2
};
```

Notice that we can also attach numerical values to our `enum` (although this is not required). To declare a variable of this type we just do the following:

```
CHOICE ch;
```

We can then set the value of this variable as follows:

```
ch = HELLO;
```

`enum` values are about making your code easier to understand. They are especially useful when working with `case` statements. Underneath, an `enum` is just an `int` value. We are simply adding some *syntactic sugar* to our code to make it easier for others to understand. A simple test application using `enum` and `case` is given below:

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

// Defines the menu choices
enum CHOICE
{
  EXIT = 0,
  HELLO = 1,
  GOODBYE = 2
};

int main(int argc, char **argv)
{
    int flag = 1;
    while (flag)
    {
        // Print menu
        printf("1 - say hello\n");
        printf("2 - say goodbye\n");
        printf("0 - exit\n");
        printf("Enter choice - ");
        // Read input
        char buffer[10];
        fgets(buffer, 10, stdin);
        // Convert to CHOICE
        enum CHOICE ch = atoi(buffer);

        // Work on input
        switch (ch)
        {
            case HELLO:
                printf("Hello World!\n");
                break;
            case GOODBYE:
                printf("Goodbye World!\n");
                break;
            case EXIT:
                printf("Exiting...\n");
                flag = 0;
                break;
            default:
                printf("*** INVALID INPUT ***\n");
                break;
        }
    }

    return 0;
}
```

Listing 2.16: Declaring and Using `enum` Values

Running this application can give you an output as follows:

```
1 - say hello
2 - say goodbye
0 - exit
Enter choice - 1
Hello World!
1 - say hello
2 - say goodbye
0 - exit
Enter choice - 2
Goodbye World!
1 - say hello
2 - say goodbye
0 - exit
Enter choice - 3
*** INVALID INPUT ***
1 - say hello
2 - say goodbye
0 - exit
Enter choice - 0
Exiting...
```

Listing 2.17: Output from Menu Application

As you can see, `enum` values are very useful for working with menu like applications. You should consider using them whenever possible to make your code easier to read.

2.10 `const`

We now know how to declare values in C. `struct` and `enum` is all C really provides in this regard. Let us now move onto examining other type modifiers. The first one we will look at is `const`. A `const` value is one that cannot be changed - it has a *constant* value.

To declare a `const` value we just put the `const` keyword before the variable type:

`const <type> <name>;`

This can be used for variable declaration and for passing parameters into functions. Below is an example program that tries to modify a `const` value.

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

// Constant global value
const int x = 500;

void func(const int i)
{
    // Print the constant
    printf("i = %d\n", i);
    // Change value of i - compiler error
    i = 50;
}

int main(int argc, char **argv)
{
    // Print the constant
    printf("x = %d\n", x);
    // Call func with 50
20     func(50);
21     // Change value of x - compiler error
22     x = 200;
23 
24     return 0;
25 }
```

Listing 2.18: Working with Constants

Trying to compile this application will give a compiler error as shown below:

```
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.30501 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

constants.c
constants.c(12) : error C2166: l-value specifies const object
constants.c(22) : error C2166: l-value specifies const object
```

Listing 2.19: Compiler Error from Trying to Modify `const` Values

If you look at the code file you will see that it is lines 12 and 22 (as noted in the compiler error) that are at fault. We are trying to modify a `const` value. The error essentially states that the left hand side of the expression (the *l-value*) is `const`.

2.11 `static` Values

Another type modifier that we can apply to our variables in C is `static`. A `static` value is one that exists throughout the execution of the application. This is especially useful for functions, where we declare variables for use in the function which only normally exist while that function is being used. `static` allows a variable to retain its value after the function has ended. A `static` value is one that has runtime context rather than just the standard context. We will look at context and variable lifespan later in the module.

For just now, try the following example application. It illustrates the basic idea of using a `static` value.

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

void func()
{
    // Define static value
    static int x = 0;

    // Print value of x
    printf("x = %d\n", x);
    // Increment x
    x = x + 1;
}

int main(int argc, char **argv)
{
    // Call func 100 times
    for (int i = 0; i < 100; ++i)
        func();

    return 0;
}
```

Listing 2.20: Using `static` Variables

Running this application provides the following output (suitably truncated):

```
x = 0
x = 1
x = 2
x = 3

....

x = 97
x = 98
x = 99
```

Listing 2.21: Output from `static` Application

Using `static` variables like this is not that common, but it can be useful occasionally. Object-orientation overcomes this requirement as we will find later.

2.12 Make Files

We will end this unit by looking at make files. A make file is a handy method of building applications by containing all the necessary instructions in a single file, rather than typing them in the command line each time.

To use a make file with Microsoft tools you have to create a file called `makefile` (note no file extension). This file contains the necessary instructions. We use a tool called `nmake` to perform the build operation for us. `nmake` looks at the `makefile` and determines which particular build we want to perform. We call `nmake` as follows:

```
nmake <what?>
```

Listing 2.22: Running the `nmake` Command

Just now, you should create a `makefile` file. Let us look at what we need to add to undertake a build.

2.12.1 Simple Make File

Below is a simple `makefile` command to build our initial `hello.c` file:

```
hello:
    cl hello.c
```

Listing 2.23: Simple makefile Example

The first line is our build we can undertake. We put a colon after the build name. After this we list the operations we want to undertake. In this instance we call `cl` on `hello.c`. Simple. Modify your `makefile` to contain these instructions and save.

We can now call `nmake` stating we want to undertake the `hello` build. We do this as follows, with the output given also.

```
nmake hello

Microsoft (R) Program Maintenance Utility Version 12.00.21005.1
Copyright (C) Microsoft Corporation.  All rights reserved.

        cl hello.c
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.30501 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

hello.c
Microsoft (R) Incremental Linker Version 12.00.30501.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:hello.exe
```

```
hello.obj
```

Listing 2.24: Building `hello.c` Using `nmake`

And it is as simple as that. You might be asking yourself *what is the point in that?*. Well, we can perform more than one instruction as part of the build, and it also allows us to store our build configurations in a file so we don't have to remember how to perform the build. This will become important as we build more complex applications. *You should start writing make files from now on.* In fact, you are required to use them for your assessment.

2.12.2 Building All

Another handy part of make files is that we can combine multiple build operations into a single build operation. This is quite common to allow us to split a build into separate parts and combine them together into a single overall build operation. We can also use it to build all the applications we have developed in one operation.

Below is an example of a `makefile` that builds all the applications by combining a collection of other builds. You should add all the necessary build configurations to your `makefile` and an `all` configuration also.

```
all: sizeof unsigned minmax datatypes

sizeof:
    cl sizeof.c

unsigned:
    cl unsigned.c

minmax:
    cl minmax.c

datatypes:
    cl datatypes.c
```

Listing 2.25: makefile Code for All

To build all we simply call `nmake` with the `all` configuration as shown below.

```
nmake all
```

Listing 2.26: Building All Applications with `nmake`

2.12.3 Cleaning Up

A final useful build configuration to have in all your make files is a clean. A `clean` configuration deletes all the files generated during your builds. This can help you tidy up any temporary files you don't need to store. A `clean` configuration is shown below:

```
clean:
    del *.obj
    del *.exe
    del *.asm
```

Listing 2.27: makefile Code for Cleaning

Make files are an important idea to understand, and you should get in the habit of creating and updating make files as you develop your applications. It will make life easier for you in the long run and it will allow you to understand the build process in more detail.

2.13 Exercises

1. Create the necessary make file(s) for all the applications you have developed thus far. Test it out to ensure that your builds work correctly, and that you have captured all the necessary configurations, including cleaning.

2. Write an application that adds 1 to the maximum value of the standard data types and subtracts 1 from the minimum. Your application should print out the values that you have after performing these operations.

3. Write an application that has a menu system using an `enum` and `case` statement which provides the ability to enter a student's details, print the student's details, or exit the system. Your application should only have one student variable in existence. It should just overwrite the existing details when a new student is entered.

4. Develop another menu based application which asks the user if they want to print out a triangle, a Christmas tree, or their name surrounded by stars. It should prompt the user for the size of the triangle / tree or their name based on the choice selected. You should use functions to simplify your application.

Unit 3

Inline Assembly

In the last unit we looked at how our C data values are represented in memory (their *low-level representation*). In this unit we will look closer at how our operations are represented. This requires us to look at assembly code. We looked at generating assembly code in the first unit. In this unit we will write some assembly code ourselves by embedding it in our C code.

Remember in the first unit we described how our compilation pipeline works. We used the following high level view:

$$\text{C code} \Rightarrow \text{Assembly code} \Rightarrow \text{Object code} \Rightarrow \text{executable}$$

In this unit we will look at how we can use assembly code directly in our C code. This can be very useful if you ever want to undertake very low-level optimisation. However, for our purposes we are trying to understand how our C code translates to assembly code. This allows us to understand our code in more detail. In general, you are unlikely to write inline assembly in all but the most rare circumstances as a C and C++ programmer. However, understanding what your code looks like on the machine is very useful.

One thing to remember is that assembly code is architecture specific. That is, the assembly code that works on a standard PC generally won't work on your mobile phone (unless it uses a Intel based CPU). Programming in assembly is very platform specific, hence the performance gains it can give.

3.1 First Inline Assembly Application

Let us look at how we add assembly code to our C code. The approach we take is supported by the Microsoft compiler, so is not guaranteed to work in other compilers. You should read the necessary documentation for these compilers to discover how to write inline assembly for them.

Declaring an Inline Assembly Block

To use inline assembly in our C code we have to declare a block of code as being assembly. For the Microsoft compiler we do this as follows:

```
__asm
{
  // Assembly commands go here
}
```

The assembly block can access any values that are accessible where the assembly block is declared. Any local variables to the function and any global variables can be accessed via their name. The C compiler will generate the proper assembly code to use them.

We won't really look at assembly code instructions supported by standard x86 processors in any detail. Table 3.1 provides an overview of the instructions we will use in this unit.

Instruction	**Parameters**	**Description**
`mov`	destination source	Moves the value stored in source to the destination. At least one of the two parameters must be a register.
`add`	destination source	Adds the value source to the value destination. The result is stored in destination. Destination needs to be a register.
`push`	source	Pushes value stored in source onto the local stack. We will discuss the stack during this unit, and look at its limitations later in the module.
`pop`	destination	Pops a value from the local stack into the destination.
`call`	procedure name	Executes the given procedure. We will look at what this means in later in the unit.
`jmp`	location	Causes control to jump to the location given. Jumping allows us to implement branching instructions like `if` and `while`.
`cmp`	value1 value2	Compares the two values. Sets relevant flags on the CPU based on the outcome of the comparison. Allows conditional jumping.
`jge`	location	A jump instruction that jumps if the result of a comparison set the greater than flag.

Table 3.1: Some Assembly Instructions

Let us now write our first application using inline assembly. Our application will declare two values (in standard C) and print out their values. Our assembly block will then store the value of one variable into the other. Finally we will print the values again (in standard C).

To store one variable into the other we use the following process:

1. Load (using `mov`) the first value into the `eax` register (the `eax` register is the standard accumulator register)
2. Load (using `mov`) the value store in the `eax` register into the second value.

And that is it. This is essentially equivalent to the following command:

```
x = y;
```

The code for this application is given below. As you can see it is very simple.

```
#include <stdio.h>

int main(int argc, char **argv)
{
  int x = 500;
  int y = 0;

  printf("Before assembly, x = %d, y = %d\n", x, y);

  __asm
  {
    mov eax, x
    mov y, eax
  }

  printf("After assembly, x = %d, y = %d\n", x, y);
  return 0;
}
```

Listing 3.1: First Application Using Inline Assembly

For this application we want to generate the assembly code again. Remember to do this we use `cl` as follows:

```
cl /c /Fa <filename.c>
```

Once your have generated your assembly code, open the `.asm` file. It should look something similar to the following.

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 18.00.30723.0

  TITLE E:\Repos\programming-fundamentals\code\unit-03\first-inline.c
  .686P
  .XMM
  include listing.inc
  .model  flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

_DATA SEGMENT
$SG3050 DB  'Before assembly, x = %d, y = %d', 0aH, 00H
  ORG $+3
$SG3051 DB  'After assembly, x = %d, y = %d', 0aH, 00H
_DATA ENDS
PUBLIC  _main
EXTRN _printf:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_x$ = -8            ; size = 4
_y$ = -4            ; size = 4
_argc$ = 8            ; size = 4
_argv$ = 12           ; size = 4
_main PROC
; File e:\repos\programming-fundamentals\code\unit-03\first-inline.c
; Line 4
  push  ebp
  mov ebp, esp
```

```
30    sub esp, 8
31  ; Line 5
32    mov DWORD PTR _x$[ebp], 500      ; 000001f4H
33  ; Line 6
34    mov DWORD PTR _y$[ebp], 0
35  ; Line 8
36    mov eax, DWORD PTR _y$[ebp]
37    push  eax
38    mov ecx, DWORD PTR _x$[ebp]
39    push  ecx
40    push  OFFSET $SG3050
41    call  _printf
42    add esp, 12          ; 0000000cH
43  ; Line 12
44    mov eax, DWORD PTR _x$[ebp]
45  ; Line 13
46    mov DWORD PTR _y$[ebp], eax
47  ; Line 16
48    mov edx, DWORD PTR _y$[ebp]
49    push  edx
50    mov eax, DWORD PTR _x$[ebp]
51    push  eax
52    push  OFFSET $SG3051
53    call  _printf
54    add esp, 12          ; 0000000cH
55  ; Line 17
56    xor eax, eax
57  ; Line 18
58    mov esp, ebp
59    pop ebp
60    ret 0
61  _main ENDP
62  _TEXT ENDS
63  END
```

Our two lines of assembly code in our C source file are represented on lines 44 and 46. Notice they aren't exactly the same as we wrote because the C compiler has generated the correct code to access the variables `x` and `y` as stored in memory.

To compile our assembly code remember we use the `ml` command. We provide a reminder below.

```
ml <filename.asm>
```

Update: Note that Microsoft no longer guarantee that you will be able to take a `.asm` file generated with `cl \Fa` and assemble it using `ml`. You may get an error from `ml`. The official line from Microsoft is that `.asm` files generated by `cl` are for information only, and are not guaranteed to be syntax compatible with the `ml` assembler. If this happens then you should just generate the executable directly using `cl` (you can still generate the `.asm` with `\Fa` to look at).

Remember your Make Files

Our build for this application has two commands - one to `cl` and one to `ml`. You should add these to your `makefile` so you can use `nmake` to build the application in one command rather than two.

Running the application provides the output below:

```
Before assembly, x = 500, y = 0
After assembly, x = 500, y = 500
```

Listing 3.2: Output from First Inline Assembly Application

This is application is really about showing you how variable assignment works in assembly code in comparison to C. There is very little difference, although it may take two assembly instructions to perform the assignment if we are assigning the value of a variable to another variable. Our next application will look at how we assign a non-variable value to a variable.

3.2 Second Inline Assembly Application

Our second application behaves very similar to the first, but now we are only going to work with one variable. This we will change the value of the single variable in the assembly block using a fixed value. Our code for this application is below.

```
#include <stdio.h>

int main(int argc, char **argv)
{
  int x = 0;

  printf("Before assembly, x = %d\n", x);

  __asm
  {
    // Same as x = 500
    mov x, 500
  }

  printf("After assembly, x = %d\n", x);

  return 0;
}
```

Listing 3.3: Second Inline Assembly Application

The assembly block is essentially the same as the following C code:

```
x = 500;
```

When generating the assembly code this time we get the following:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 18.00.30501.0

  TITLE D:\programming-fundamentals\code\unit-03\second-inline.c
  .686P
  .XMM
  include listing.inc
  .model  flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

_DATA SEGMENT
$SG3049 DB  'Before assembly, x = %d', 0aH, 00H
  ORG $+3
$SG3050 DB  'After assembly, x = %d', 0aH, 00H
```

```
16 _DATA ENDS
17 PUBLIC  _main
18 EXTRN _printf:PROC
19 ; Function compile flags: /Odtp
20 _TEXT SEGMENT
21 _x$ = -4            ; size = 4
22 _argc$ = 8            ; size = 4
23 _argv$ = 12           ; size = 4
24 _main PROC
25 ; File d:\programming-fundamentals\code\unit-03\second-inline.c
26 ; Line 4
27   push  ebp
28   mov ebp, esp
29   push  ecx
30 ; Line 5
31   mov DWORD PTR _x$[ebp], 0
32 ; Line 7
33   mov eax, DWORD PTR _x$[ebp]
34   push  eax
35   push  OFFSET $SG3049
36   call  _printf
37   add esp, 8
38 ; Line 12
39   mov DWORD PTR _x$[ebp], 500     ; 000001f4H
40 ; Line 15
41   mov ecx, DWORD PTR _x$[ebp]
42   push  ecx
43   push  OFFSET $SG3050
44   call  _printf
45   add esp, 8
46 ; Line 17
47   xor eax, eax
48 ; Line 18
49   mov esp, ebp
50   pop ebp
51   ret 0
52 _main ENDP
53 _TEXT ENDS
54 END
```

Listing 3.4: Assembly Code Generated from Second Inline Assembly Application

Notice the comment on line 39 - `000001f4H`. This is the value 500 in hexadecimal (the H denotes this). Try it out on a calculator - convert 1f4 to decimal. Running this application will provide the following output:

```
Before assembly, x = 0
After assembly, x = 500
```

Listing 3.5: Output from Second Inline Assembly Application

We have now looked at variable assignment in assembly, and you can see how C commands roughly translate to assembly code. Let us now move on to other operations.

3.3 Using Assembler Operations

Assembly provides a number of standard operations to manipulate numerical values. One of these is `add`. This command allows us to add a value to one of the registers.

This is our standard method of adding values together. To do this we normally perform the following operations in assembly:

1. Store first value into a register
2. Add second value to the register
3. Store the result to memory

This is the standard approach to performing simple arithmetic operations in assembly code. There a number of different assembly code instructions that deal with arithmetic operations. Our application will perform addition.

Our application will add two variables - `x` and `y` - and store the result in another variable `z`. We will use the `ecx` register to undertake our calculation. Converting our steps above into assembly code we get the following:

1. `mov ecx, x`
2. `add ecx, y`
3. `mov z, ecx`

Our application code is below:

```
 1 #include <stdio.h>
 2 
 3 int main(int argc, char **argv)
 4 {
 5   int x = 500;
 6   int y = 200;
 7   int z = 0;
 8 
 9   // Add using assembly
10   __asm
11   {
12     // Move x into ecx register
13     mov ecx, x
14     // Add y to ecx register
15     add ecx, y
16     // Store in z
17     mov z, ecx
18   }
19 
20   printf("z = %d\n", z);
21 
22   // Add in code
23   z = x + y;
24 
25   return 0;
26 }
```

Listing 3.6: Adding Two Values in Assembly

The application is the equivalent of the following code. In fact we also perform the operation at the end of the main application (line 23). You can examine this as part of an exercise.

```
z = x + y;
```

Running this application will provide you with the following output:

```
z = 700
```

Listing 3.7: Output from Adding Two Values in Assembly

3.3.1 Exercises

1. Try some other arithmetic operations and test the results. Ensure that you get the result you expected. The operations to try are:

    - `sub` - subtraction
    - `imul` - multiplication

    We won't try division here as it is a bit more complicated, but the recommended reading will help.

2. For each application you have built in this section (addition, subtraction, multiplication) compare the resulting Assembly code you have written to the one generated by the compiler (such as the command on line 23 in the last example). Ensure you are getting the same result to ensure you understand the concepts covered.

3. Are you still writing make files? We hope so - it makes your life easier.

3.4 Using the Stack

Now that we understand the basic concepts of how simple instructions in C are converted to Assembly code instructions we can move onto starting to understand what happens when we call functions in C. You might have seen some of the code for this in our Assembly code generated by the compiler, but we haven't really gone into any detail yet. Before we can really look at function calling, we have to understand a fundamental part of our running application - *the stack*.

3.4.1 What is the Stack?

When working with a running application written in a low level language such as C, we have to consider two parts of memory. The first part - and the only part we have really been working in - is the stack. The other part of memory, *the heap*, we will cover in more detail later in the module when we look at memory management.

The stack can be considered the *working memory* of our application. It keeps track of the variables that are currently in *scope*. Variables in scope are the values that we can currently directly access. Again, we will cover exactly what this means in a later unit. However, for the moment you should understand that when you call a function that only variables passed into the function are accessible, unless you have any global variables as well.

The stack is therefore a list of the values that you have created during the running of your application using standard variable declaration. It allows us to grab these variables and use them for our operations. The CPU keeps track of the current stack for us, adding or removing values as they come into and out of scope.

As an example, consider the following piece of code:

```
void func()
{
  // Enter the function
  // Declare 2 values
  int x = 500;
  int y = 1000;
  {
    // Enter an inner scope
    int z = x + y;
    // Exit inner scope - z removed from stack
  }
  // Exit the function - x and y removed from stack
}
```

The code above operates on the stack as illustrated in Figure 3.1. When we enter the function, we can consider the stack to be empty (it isn't really, but for our example here we can consider it as such). When we declare our first variable `x` it is added to the stack. Then `y` and finally `z`. As each scope is exited, certain values are removed from the stack. Therefore, we keep track of values that have been declared.

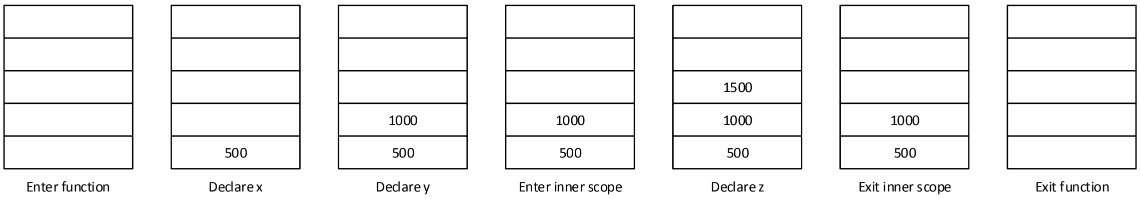


Figure 3.1: Values Added and Removed from the Stack During Operation

WARNING - This is a Simplified Idea of the Stack

This is not truly how the stack works in practice. Space is allocated on the stack well before you actually declare the values. The stack is in fact a limited resource. Generally only 1 MB of space is allocated to a stack, and we have an offset on the stack to the various variables that have been declared. A *stack pointer* keeps track of how much of the stack is in scope, moving up and down based on need. No values are really *removed* from the stack, the CPU just considers memory beyond the stack pointer as no longer valid and may overwrite it if it wishes.

We will see why we need the stack for this purpose through this unit of the module and some later units. For just now, we will look at how we work with the stack.

3.4.2 Working with the Stack

When it comes to working with the stack, we are really only interested in the following two commands:

- `push` - adds a value to the stack
- `pop` - removes a value from the stack and stores it in the given location

These two machine instructions are very simple, and only take a single parameter. For example, if we want to store the value in variable `x` on the stack, we use the command `push x`. To remove a value from the stack and store it in variable `y` we use `pop y`. The instructions are simple, *but the idea of how the stack operates is important.* You should familiarise yourself with the basic concept, and use the next few examples to fully understand what is happening.

3.4.3 Test Application

Our test application will swap two variables - `x` and `y` - using the stack as a swap space. We already know that this is not how the C compiler would actually perform this operation, so do not convince yourself that it is. Therefore this isn't a real world example. However, this example does give you a good idea of how the stack operates.

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  int x = 500;
  int y = 200;

  __asm
  {
    // Push x onto the stack
    push x
    // Push y onto the stack
    push y
    // Pop stack into x
    pop x
    // Pop stack into y
    pop y
  }

  printf("x = %d, y = %d\n", x, y);

  return 0;
}
```

Listing 3.8: Using the Stack to Swap Values

Figure 3.2 illustrates how this application operates when executing the Assembly code section (although the stack isn't really empty). The output of this application is also shown below.

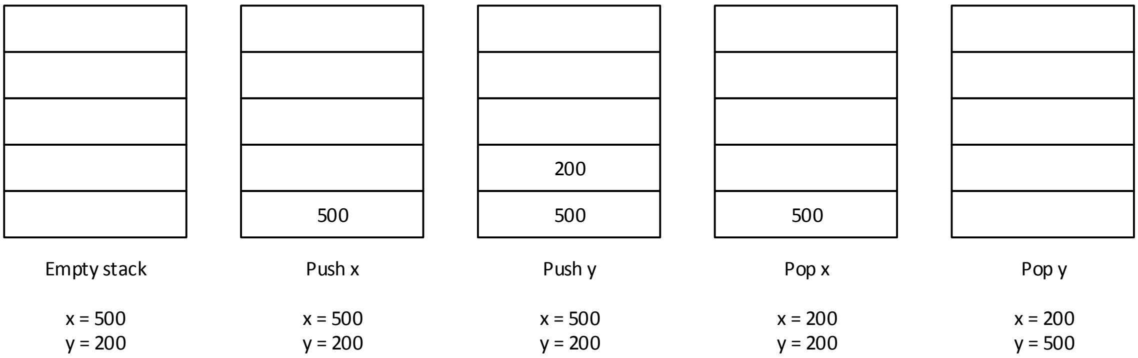


Figure 3.2: Using the Stack to Swap Values with `push` and `pop`

```
x = 200, y = 500
```

Listing 3.9: Output from Stack Swapping Application

3.5 Calling Functions - `printf`

With our understanding of the stack we can now move onto calling actual functions. This requires us to set parameters to pass to the function - which is where the stack comes in. However, before moving onto how we use the stack to accomplish this, we will look at how we actually *call* a function.

3.5.1 Calling Procedures in Assembly

To call a procedure in Assembly we use the `call` command. All this instruction requires is the name of the procedure you wish to call. For example, to call `printf` we simply use the instruction `call printf`. This will cause the CPU to *jump* to the set of instructions that define the `printf` procedure. At the end of the procedure, the CPU returns to where it encountered the `call` operation and continues executing the instructions. This is achieved by the `ret` instruction.

3.5.2 Setting the Stack

As you should realise by now, calling a function / procedure is only one part of the puzzle. We also have to pass parameters into the call for the procedure to use. This is where the stack comes in.

Before calling a procedure, you have to add the values of the parameters to the stack. This is because the called procedure needs a *copy* of these values (we will look at call conventions later in the module). As we need to create a copy, and the procedure needs to know where these values are stored (based on it's parameter list), we have to push these values onto the stack.

As an example, consider a function defined as follows:

```
void func(int x, int y) ...
```

We need to set the values of `x` and `y` for the procedure to operate. For example, let us say we called the procedure as follows:

```
func(1000, 2000);
```

To actually call this function in assembly we would have to use the following instructions:

```
push 2000
push 1000
call func
```

Notice that we push the parameter values from right to left, not left to right. This is because the procedure will look at the top of the stack as the starting point for its parameters and work downwards. Therefore, the first parameter for the procedure (`x` in our example) has to be at the top of the stack, the second one value down, the third two values down, etc. *This is another important idea to realise and is underpinned by an understanding of how the stack works - last-in, first-out.*

3.5.3 Clearing the Stack

Setting the stack for use by a procedure is what you should do before you call it. After the procedure has returned you need to clean up the stack again. This is done by popping the relevant number of values of the stack (actually in practice there is a better way - see the exercises). In our example we will pop any stack values into the `ebp` register to clean the stack.

3.5.4 Example Application

To review, we use the following three stage process when calling a procedure in Assembly code:

1. Set the stack with the relevant parameters. Remember we push these values in reverse order than we use them.
2. Call the procedure.
3. Clean the stack of any set parameters.

The following application illustrates this basic process by returning to our simple *Hello World* example, but this time using inline assembly to perform the actual call.

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  char *message = "Hello World!\n";

  __asm
  {
    // Push message onto the stack
    push message
    // Call printf
    call printf
    // Clean up stack - pop into ebx
    pop ebx
  }

  return 0;
}
```

Listing 3.10: Calling `printf` From Assembly

Compile this code and generate the Assembly. This is given below:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 18.00.30501.0

  TITLE D:\programming-fundamentals\code\unit-03\call.c
  .686P
  .XMM
  include listing.inc
  .model  flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

_DATA SEGMENT
$SG3049 DB  'Hello World!', 0aH, 00H
```

```
14 _DATA ENDS
15 PUBLIC  _main
16 EXTRN _printf:PROC
17 ; Function compile flags: /Odtp
18 _TEXT SEGMENT
19 _message$ = -4            ; size = 4
20 _argc$ = 8            ; size = 4
21 _argv$ = 12           ; size = 4
22 _main PROC
23 ; File d:\programming-fundamentals\code\unit-03\call.c
24 ; Line 4
25   push  ebp
26   mov ebp, esp
27   push  ecx
28   push  ebx
29 ; Line 5
30   mov DWORD PTR _message$[ebp], OFFSET $SG3049
31 ; Line 10
32   push  DWORD PTR _message$[ebp]
33 ; Line 12
34   call  _printf
35 ; Line 14
36   pop ebx
37 ; Line 17
38   xor eax, eax
39 ; Line 18
40   pop ebx
41   mov esp, ebp
42   pop ebp
43   ret 0
44 _main ENDP
45 _TEXT ENDS
46 END
```

Listing 3.11: Assembly Code from Calling `printf` Using Assembly

There are a few lines we will look at in this example:

- On line 16 we see the declaration for the `_printf procedure` (the compiler adds an underscore before procedure names to get round keyword usage issues). Notice the use of the term `EXTRN`. This declares the `printf` function as *external* to the Assembly code we are compiling. This is because `printf` is stored in a library. We will look at these ideas in the next lesson.

- Line 32 is where the parameter is set in Assembly code. Here we are providing a memory location to `printf`. Remember how our strings are stored.

- Line 36 is where our stack cleaning occurs.

Running this application will display the following:

```
Hello World!
```

Listing 3.12: Output from Assembly Hello World Application

Not much different than the previous *Hello World* example in pure C. However, you now know the Assembly code that the C compiler has generated.

3.5.5 Exercise

Modify the inline assembly so that you call `printf` multiple times, but do not clean the stack after each call (no popping int `ebx`)

3.6 Calling Functions - Writing your Own

Let us now see what happens when we create and call our own function in C. To do this we will write a small subtraction function that will take two values as parameters and return the difference. The code for this sample is given below:

Getting Return Values from a Function Call

So far we haven't covered how we get our return value after a function call. Well, this value has been stored for us in one of the registers - `eax`. This means that we can retrieve it from there if we wish. For example, let us say we are going to write the following code:

```
int z = sub(1000, 500);
```

In Assembly code this becomes the following set of instructions (if we do naive stack cleaning):

```
1 push 500
2 push 1000
3 call sub
4 mov z, eax
5 pop ebx
6 pop ebx
```

That line 4 is the important one. This is where we store the result of the function call into our required memory location. It is the job of the function to store the function result into the `eax` register.

```
#include <stdio.h>

int sub(int x, int y)
{
  return x - y;
}

int main(int argc, char **argv)
{
  int result = 0;

  __asm
  {
    // Push 500 onto stack
    push 500
    // Push eax onto stack
    push 200
    // Call sub
    call sub
    // Move eax into result
    // eax contains the result of the call
    mov result, eax
    // Clean up stack - pop into ebx
    pop ebx
    pop ebx
  }

  printf("result = %d\n", result);

  return 0;
}
```

Listing 3.13: Calling a User Defined Function

You should understand this code enough so that we don't have to explain it in any detail. The more important part if the generated assembly code which is below:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 18.00.30501.0

  TITLE D:\programming-fundamentals\code\unit-03\call2.c
  .686P
  .XMM
  include listing.inc
  .model  flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

_DATA SEGMENT
$SG3055 DB  'result = %d', 0aH, 00H
_DATA ENDS
PUBLIC  _sub
PUBLIC  _main
EXTRN _printf:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_result$ = -4           ; size = 4
_argc$ = 8            ; size = 4
_argv$ = 12           ; size = 4
_main PROC
; File d:\programming-fundamentals\code\unit-03\call2.c
; Line 9
  push  ebp
  mov ebp, esp
  push  ecx
  push  ebx
; Line 10
  mov DWORD PTR _result$[ebp], 0
; Line 15
  push  500         ; 000001f4H
; Line 17
  push  200         ; 000000c8H
; Line 19
  call  _sub
; Line 22
  mov DWORD PTR _result$[ebp], eax
; Line 24
  pop ebx
; Line 25
  pop ebx
; Line 28
  mov eax, DWORD PTR _result$[ebp]
  push  eax
```

```
47   push  OFFSET $SG3055
48   call  _printf
49   add esp, 8
50 ; Line 30
51   xor eax, eax
52 ; Line 31
53   pop ebx
54   mov esp, ebp
55   pop ebp
56   ret 0
57 _main ENDP
58 _TEXT ENDS
59 ; Function compile flags: /Odtp
60 _TEXT SEGMENT
61 _x$ = 8                ; size = 4
62 _y$ = 12               ; size = 4
63 _sub  PROC
64 ; File d:\programming-fundamentals\code\unit-03\call2.c
65 ; Line 4
66   push  ebp
67   mov ebp, esp
68 ; Line 5
69   mov eax, DWORD PTR _x$[ebp]
70   sub eax, DWORD PTR _y$[ebp]
71 ; Line 6
72   pop ebp
73   ret 0
74 _sub  ENDP
75 _TEXT ENDS
76 END
```

Listing 3.14: Assembly Code from User Defined Function

The lines of interest here are as follows:

- Line 15 has the declaration of our `sub` function. Notice this time that it is declared as `PUBLIC` rather than `EXTRN`. This indicates that it is a publicly callable function in this code.
- Lines 33 and 35 set our stack. Remember we are pushing the values in reverse order. This means that `x` is 200 and `y` is 500.
- Line 37 issues the `call` instruction.
- Line 39 is where we retrieve the return value of the function call from the `eax` register and store it in `z`.
- Lines 41 and 43 clean up the stack.
- Line 63 is where the `sub` procedure is defined
- Lines 69 and 70 is where the actual calculation takes place. Notice we use the `eax` register. This is where the result is stored.
- Line 73 is where the procedure returns, going back to line 39 to continue execution.

Running this application gives the following result:

```
result = -300
```

Listing 3.15: Output from Calling User Defined Function in Assembly

3.7 For the Brave - Loops

OK, this is the first section that is meant for people who really want to push their understanding of the work we are doing in this module. The *For the Brave* sections are not necessary to complete the module assessment, but they will help you really understand the concepts covered and let you explore some more advanced issues.

In this section we will look at how we work with branching statements to build loop like code in Assembly. To do this we will recreate our application that prints out the command line arguments. This requires us to use three new Assembly instructions:

1. `jmp` - causes the CPU to jump to the instruction at the given label.
2. `cmp` - compares two values and sets some flag values on the CPU based on the comparison.
3. `jge` - causes the CPU to jump to the instruction at the given label if the greater than or equal to flags were set after a comparison.

The comparison and jump instructions are the basis of allowing us to build any branching statement in Assembly. This takes a bit more work than standard assignment and function calling Assembly code. However, the basic premise of converting C code to assembly still remains.

Labels and Jumping

In Assembly we have to place labels in our code and tell the CPU to jump to these instructions when we need it to. This is how branching works on the CPU. However, you may have been warned about writing such code in other programming classes. This is the classic `goto` statement idea, which leads to *spaghetti code*. C (and by extension C++) provide a `goto` statement - *and you should never use it!*. `goto` as a command has its uses but you will never need it for anything but very, very particular purposes. It is hard to debug and hard to track how your application works. Let the compiler create any necessary labels and jumps. You should avoid it.

For our application, the Algorithm 6 provides an outline of the operations to occur.

The code we are going to use is as follows:

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

int main(int argc, char **argv)
{
    // Format string for message
    char *format = "Argument %d: %s\n";
    // Final message
    char *message = "All arguments printed.\n";
    // Iteration value
    int i;
    __asm
    {
        // Set i to 0
        mov i, 0
```

Algorithm 6 A Loop Application to Display Command Arguments Written in Assembly

```
procedure ASSEMBLY LOOP
    $i \leftarrow 0$
    goto CHECK_i
                                                    ▷ INCREMENT_i
    $eax \leftarrow i$
    $eax \leftarrow eax + 1$
    $i \leftarrow eax$
                                                    ▷ CHECK_i
    $ecx \leftarrow i$
    Compare $ecx$ to $argc$
    if greater than then goto END_LOOP
    $edx \leftarrow i$
    $eax \leftarrow argv$
    $ecx \leftarrow eax + edx \times 4$
    push $ecx$ onto the stack
    push $i$ onto the stack
    push $format - message$ onto the stack PRINTF
    pop the stack into $ebx$
    pop the stack into $ebx$
    pop the stack into $ebx$
    goto INCREMENT_i
                                                    ▷ END_LOOP
    push $end - message$ onto the stack PRINTF
    pop the stack into $ebx$
```

```
        // Jump to check
        jmp $CHECK_i

    // This section of code increments i
    // Look up the inc instruction - simplify your code
    $INCREMENT_i:
        mov eax, i
        add eax, 1
        mov i, eax

    // This section of code checks if i < argc
    $CHECK_i:
        mov ecx, i
        cmp ecx, argc
        jge $END_LOOP

        // This is the body of the loop
        mov edx, i
        mov eax, argv
        // This line takes the base address of argv and adds 4 * i bytes further along
        // Remember that a memory address in 32-bit applications is 4 bytes
        mov ecx, [eax + (edx * 4)]
        // Push the arguments onto the stack
        push ecx
        push i
```

```
41          push format
42          // Call printf
43          call printf
44          // Pop data from the stack to clear it
45          pop ebx
46          pop ebx
47          pop ebx
48
49          // Increment i
50          jmp $INCREMENT_i
51
52      // This is called after the end of the loop
53      $END_LOOP:
54          push message
55          call printf
56          pop ebx
57      }
58
59      return 0;
60  }
```

Listing 3.16: Looping through the Command Line Arguments with Inline Assembly

This follows a classic *sequence-selection-iteration* style. This is all the CPU can do. High level languages allow us to abstract further and further from the low level code representation understood by the CPU to instructions that are understandable by the programmer. Notice the work put in to allow us to implement a loop in Assembly against our C code.

Running this application gives us an output similar to the following (depending on the instructions passed as shown on line 1):

```
loops hello world
Argument 0: loops
Argument 1: hello
Argument 2: world
All arguments printed.
```

Listing 3.17: Output from Printing Command Line Arguments Using Inline Assembly

3.7.1 Exercise

Modify the code on lines 22 to 24 to use the Assembly instruction `inc` instead. You should find this fairly easy.

3.8 Why did we Just Look at Assembly?

Just to end this unit, we will reflect on why we have looked at Assembly. The main point here is to understand how C code is converted into Assembly code. It is rare that you would write the type of Assembly code we have, but hopefully you see the similarity between standard C code and what the machine executes as instructions. Remember that a typical processor these days will have a clock speed between 2 and 4 GHz. This implies that the CPU is executing 2 to 4 million of these instructions per second (again, this is a simplified description - it isn't as easy as that). The fact that C compiles to a small, compact Assembly instructions is the reason C is considered a fast language.

3.9 Additional Resources

Our aim in this unit was to give you an understanding of how C code converts to Assembly. As such, we haven't delved too deeply into a wide range of aspects in low level programming. However, if you are interested the following link provides a reasonable starting position:

`http://www.cs.virginia.edu/~evans/cs216/guides/x86.html`

3.10 Exercises

1. Still using make files? Make sure that you are!
2. Investigate the `struct` examples from the previous unit. You should generate the Assembly code from some of the previous examples and examine how C works with `struct` values. Try your best to try and figure out what is happening, but don't let yourself get to caught up if you are unsure. Ask for help if needed.
3. Using the `esp` register. The `esp` register is the stack pointer, and can be modified to tell the CPU where it should consider the stack to be valid. When we looked at function calling, we used `pop` to clear the values we added to the stack. This is efficient for one parameter, but not the most efficient for greater numbers of parameters (lots of `pop` instruction). A better method is to set the `esp` register so that it points back to where the valid stack ends. Can you change some of our example applications to achieve this? *Hint - the Assembly code from some of the other applications where you have called functions may help you here.* The `ebp` register (*stack base pointer*) is also used.
4. **For the Brave** - Write an application, using inline assembly as much as possible, that prompts the user for their name and reads in the result using `fgets`. This is not as simple as it sounds.
5. **For the Brave** - Write an application, using inline assembly as much as possible, that works on an array of 10 integers. The application will have two `for` loops. The first loop will set the value of the numbers in the array from 0 to 9. The second `for` loop will print the entries and sum the total. At the end of the second `for` loop the total should be displayed.

Unit 4

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*.

4.1 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.

4.1.1 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 4.1 provides some examples.

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

Listing 4.1: 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 4.2 is an example application using #define before the pre-processor is run across it.

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

Listing 4.2: 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 4.3.

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

Listing 4.3: 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 4.4.

```
#ifdef TEST
printf("Test defined\n");
#else
printf("Test not defined\n");
#endif
```

Listing 4.4: 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 4.1 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 4.2 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 4.1: 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 4.2: Some Pre-Processor Commands

4.1.2 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.

```
cl /D <value> <filename.c>
```

Listing 4.5: 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:

```
cl /D DEBUG 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.

4.1.3 Using `#ifdef` for Conditional Compilation

OK, let us now put what we have learnt to the test. Listing 4.6 is our example code that you should enter using Notepad++.

```
#include <stdio.h>

int main(int argc, char **argv)
```

```
{
#ifdef DEBUG
  printf("In debug mode\n");
#elif RELEASE
  printf("In release mode\n");
#else
  printf("What mode am I in?\n");
#endif
  return 0;
}
```

Listing 4.6: Using `#ifdef`

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

4.1.4 Exercise

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

4.2 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.

```
#pragma once
// This file needs to know what printf is
#include <stdio.h>

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

Listing 4.7: `hello.h` Header File

What is `#pragma once`?

We already mention in Table 4.2 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:

```c
#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.

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

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

Listing 4.8: 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 `cl` 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:

```c
#include <stdio.h>

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

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

Listing 4.9: 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 Microsoft C library this file is over 700 lines of code long. Hence we haven't included it here.

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.

4.3 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.

4.3.1 Compiling Multiple Files

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

```
cl <filename1.c> <filename2.c> <filename3.c> ....
```

Listing 4.10: 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 the name of the first file used in the compile. For example, compiling with:

```
cl hello.c goodbye.c
```

will produce an executable called `hello.exe`. We can control the name of the executable by using the `/Fe` flag with the compiler. For example, to control the set the name of the executable above to `myapp.exe` we would use the following:

```
cl /Fe:myapp.exe 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 4.1 gives you the general idea of how headers and header inclusion creates this bridge.

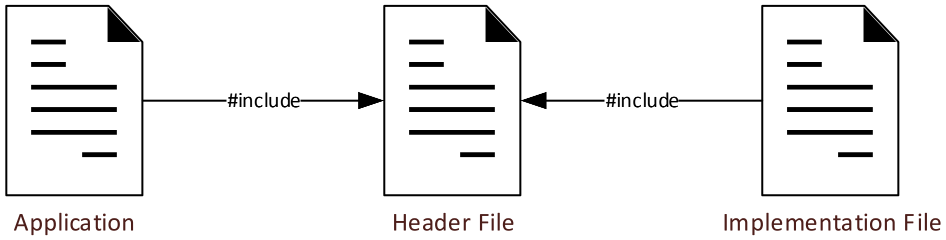


Figure 4.1: Inclusion Structure

4.3.2 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:

```c
#pragma once

// A structure representing a student
struct student
{
    unsigned int matric;
    char *name;
    char *address;
};

// Declaration of print student method - not implementation
void print_student(struct student s);
```

Listing 4.11: `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:

```c
#include "student.h"
#include <stdio.h>

void print_student(struct student s)
{
  printf("Matric: %d\n", s.matric);
  printf("Name: %s\n", s.name);
  printf("Address: %s\n", s.address);
}
```

Listing 4.12: `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:

```c
#include "student.h"

int main(int argc, char **argv)
{
  struct student s;
  s.matric = 123456;
  s.name = "Kevin Chalmers";
  s.address = "Edinburgh Napier University";
  print_student(s);

  return 0;
}
```

Listing 4.13: 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 4.2. 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 `obj` files are linked together to form the main application.

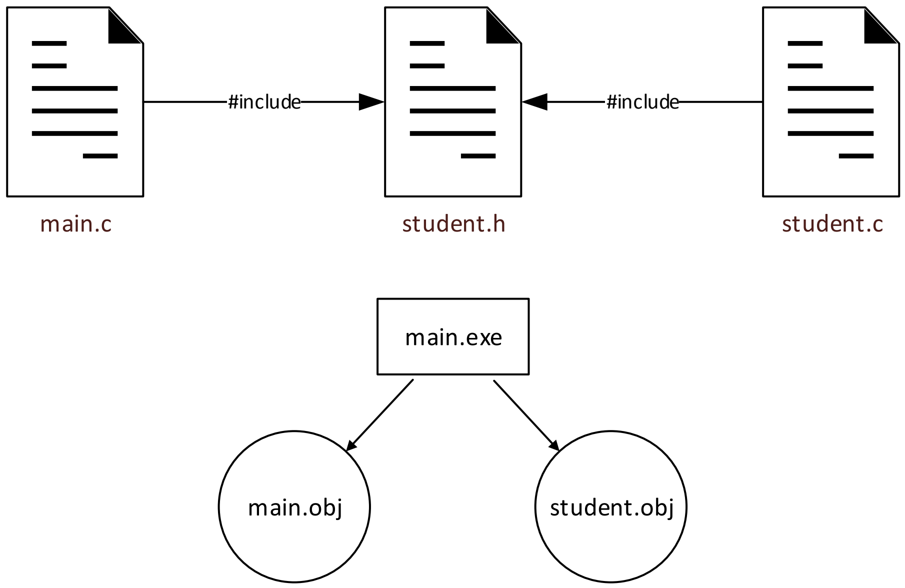


Figure 4.2: Structure of Student Application - both File Inclusion and Object Files

4.3.3 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.

4.4 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 Microsoft compiler tools.

4.4.1 Compiling Code into a Library

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

```
cl /c <filename1.c> <filename2.c> <filename3.c> ...
// Compile as many files as necessary into object files
lib <filename1.obj> <filename2.obj> <filename3.obj> ...
// Reusable library of code generated
```

Listing 4.14: Creating a Library from Object Files

Our header files will still act as a bridge for us as shown in Figure 4.3. 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.

4.4.2 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 `link` command). 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.

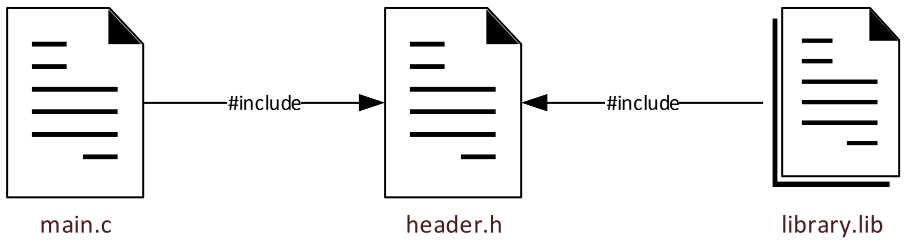


Figure 4.3: Library Inclusion Structure

```
link <filename.obj> <lib-name1.lib> <lib-name2.lib> ...
```

Listing 4.15: Linking to Libraries

This means that our file link would look like that shown in Figure 4.4. It is essentially the same idea as when linking object files in general.

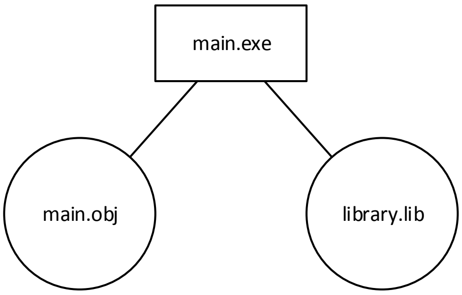


Figure 4.4: Linking a Library

4.4.3 Exercise - Student Details (Again)

Your task this time is to compile the `student.c` file into a library (which will end up being called `student.lib`). Once that library is created, create the main application by linking the main `.obj` file to the `student.lib` file.

4.5 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.

4.5.1 `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.

```c
#pragma once

int search(int value, int size, int *data);
```

Listing 4.16: Search Header File

4.5.2 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 7 provides the pseudocode for the linear search algorithm.

Algorithm 7 Linear Search Algorithm

```
function SEARCH($value$, $size$, $data$)
    for $i \leftarrow 0$ to $size - 1$ do
        if $data_i = value$ then
            return $i$
    return $-1$
```

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

```c
#include "search.h"

int search(int value, int size, int *data)
{
  // Loop throught data until found
  for (int i = 0; i < size; ++i)
  {
    if (data[i] == value)
    {
      // Found value - return i
      return i;
    }
  }
  // Not found.  Return -1
  return -1;
}
```

Listing 4.17: Main Search File

4.5.3 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.

```c
#pragma once

void sort(int size, int *data);
```

Listing 4.18: Sort Header File

4.5.4 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 8 provides the pseudocode for this capability.

Algorithm 8 Bubble Sort Algorithm

```
procedure SORT($size$, $data$)
    for $i \leftarrow 0$ to $size - 1$ do
        for $j \leftarrow 0$ to $size - (i + 1)$ do
            if $data_j < data_{j+1}$ then
                Swap values
```

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

```c
#include "sort.h"
#include <stdio.h>

void sort(int size, int *data)
{
  // Iterate through each value
  for (int i = 0; i < size; ++i)
  {
    // Loop through values above index i
    for (int j = 0; j < size - (i + 1); ++j)
    {
      // Test if data[j] > data[j + 1]
      if (data[j] > data[j + 1])
      {
        // Swap values
        int temp = data[j + 1];
        data[j + 1] = data[j];
        data[j] = temp;
      }
    }
    // Display % of currently sorted data
    if (i % 1000 == 0)
      printf("%.2f%% sorted\n", ((float)i / (float)size) * 100.0f);
  }
}
```

Listing 4.19: 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.

4.5.5 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.

```c
#pragma once

void generate(int size, int *data);
```

4.5.6 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.

```c
#include "generate.h"
#include <stdlib.h>
#include <time.h>

void generate(int size, int *data)
{
  // Seed the random
  srand(time(NULL));
  // Generate random numbers
  for (int i = 0; i < size; ++i)
    data[i] = rand();
}
```

Listing 4.20: 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.

4.5.7 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`.

```c
#include <stdlib.h>
#include <stdio.h>
#include "generate.h"
#include "sort.h"
#include "search.h"

#define NUM_INTEGERS 65535

int main(int argc, char **argv)
{
  // Allocate an array of a given size
  int data[NUM_INTEGERS];

  // Generate random numbers
  generate(NUM_INTEGERS, data);

  // Output first 20 values
  printf("\nUnsorted\n");
  for (int i = 0; i < 20; ++i)
    printf("%d\n", data[i]);

  // Sort the data
  sort(NUM_INTEGERS, data);

  // Output first 20 values
  printf("\nSorted\n");
  for (int i = 0; i < 20; ++i)
    printf("%d\n", data[i]);

  return 0;
}
```

Listing 4.21: Test Application for Array Library

Our inclusion file structure is shown in Figure 4.5.

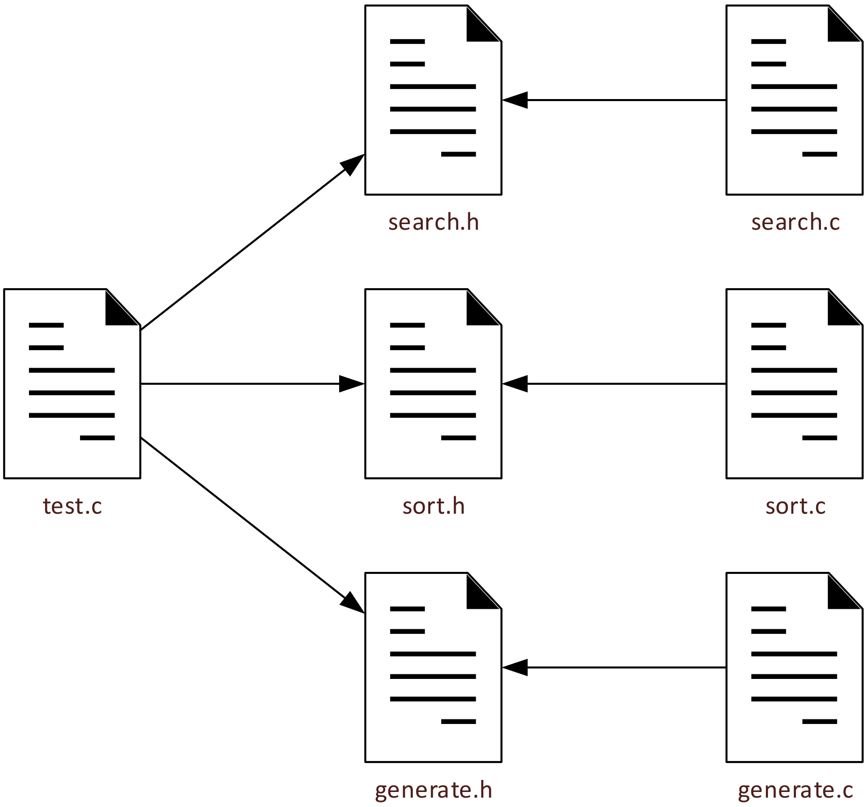


Figure 4.5: Array Library File Structure

4.5.8 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!

```
cl /c search.c sort.c generate.c
lib /OUT:array.lib search.obj sort.obj generate.obj
cl /c test.c
link test.obj array.lib
```

Listing 4.22: Command Line Arguments to Compile the Array Library

Defining the Library Name

As with the `cl` command we can also define the library name using the `lib` command. This is also done using the `/OUT` flag. The commands show us doing this to create a file called `array.lib`

4.5.9 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 4.23: 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.

4.6 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.

4.6.1 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.

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

Listing 4.24: 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 4.3 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 4.3: File Opening Modes in C

4.6.2 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:

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

Listing 4.25: 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.

4.6.3 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.

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

// Reads in a block of data as an int array
int readfile(int **data)
{
    // Open file for reading
    FILE *file;
    file = fopen("numbers.dat", "rb");
    // First value is number of integers
    int size;
    fread(&size, sizeof(int), 1, file);
    // Allocate memory for that number of values
    *data = (int*)malloc(sizeof(int) * size);
    // Read in rest of data
    fread(*data, sizeof(int), size, file);
    // Return size
    return size;
}

int main(int argc, char **argv)
{
    // Declare data
    int *data;
    // Read in file
    int size = readfile(&data);
    // Sort
    sort(size, data);
    // Print first 20 results
    for (int i = 0; i < 20; ++i)
        printf("%d\n", data[i]);

    // Free the allocated memory - otherwise a memory leak! (very bad)
    free(data);

    return 0;
}
```

Listing 4.26: 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.

4.6.4 Exercise

You should be able to compile and link this file. You will need the array.lib 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.

4.7 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.

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

// Reads in a block of data as an int array
int readfile(int **data)
{
    // Open file for reading
    FILE *file;
    file = fopen("numbers.dat", "rb");
    // First value is number of integers
    int size;
    fread(&size, sizeof(int), 1, file);
    // Allocate memory for that number of values
    *data = (int*)malloc(sizeof(int) * size);
    // Read in rest of data
    fread(*data, sizeof(int), size, file);
    // Close file
    fclose(file);
    // Return size
    return size;
}

// Writes strings to the file
void writefile(int size, int *data)
{
    // Create file
    FILE *file;
    file = fopen("sorted.txt", "w");
    // Loop through each value, writing to the file
    for (int i = 0; i < size; ++i)
        fprintf(file, "%d\n", data[i]);
    // Close the file
    fclose(file);
}

int main(int argc, char **argv)
{
    // Read in data
    int *data;
    int size = readfile(&data);
    // Sort
    sort(size, data);
    // Write the data
    writefile(size, data);
    // Free the allocated memory - otherwise a memory leak! (very bad)
    free(data);

    return 0;
}
```

Listing 4.27: 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).

4.8 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 5

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++. This is the language we will use from now on.

5.1 The C++ Programming Language

C++ was first developed in the late 1970s / early 1980s. It is considered a successor or the next step (hence the ++) to C. As such, everything you have done up until this moment in C works in C++. C++ just adds some new constructs to make our life *a lot* easier. One of the major differences between C and C++ is that C++ has *object-orientation* (like Java and C#). We won't look at object-orientation in C++ until a few units, but we already introduced some of the basic ideas with C `structs`.

As with everything, describing a programme language doesn't tell you as much as actually using the language. With that in mind let us move onto a simple *Hello World* example. *From this point onwards you should consider that every piece of code we provide as an example is C++.*

5.2 Hello World in C++

Our *Hello World* example is given below. The extension of a C++ file is `.cpp`. Otherwise, we use the `cl` compiler as normal. Therefore, you should save the following code as `hello.cpp` and compile it using `cl hello.cpp`.

```
1 #include <iostream>
2 
3 int main(int argc, char **argv)
4 {
5     std::cout << "Hello World!" << std::endl;
6 
7     return 0;
8 }
```

Listing 5.1: Hello World C++

OK, we have a couple of new ideas here. First, the include file. Here we are including the file `iostream` (note no `.h`). This is part of the standard C++ library and provides some basic input-output mechanisms (such as command line input-output). The header name is an abbreviation of *input-output streams*.

On line 5 we have our *print* statement - and this will look very strange to new C++ programmers. First of all, let us describe the two objects being used:

`std::cout` - this is the command line. It is a member of the *Standard Library* namespace (hence the `std::`). *cout* can be interpreted as *Console OUTput*.

`std::endl` - this is a new line character supported by the operating system. It is a member of the *Standard Library* namespace (hence the `std::`). *endl* can be interpreted as *END Line*.

Now the new operators. We are using the operator `<<` which is known as the `stream output` operator. You can read the entire command as *to the console stream "Hello World!" then an end line*. There is an equivalent *stream input* operator (`>>`) which you should look at in an exercise soon.

Compiling and running this application should give you the expected output:

```
Hello World!
```

Listing 5.2: Output from C++ Hello World Application

5.3 Working with `std::string` in C++

We spent a fair bit of time working with character strings in C. In particular, we looked at how strings are represented in C, how this can cause issues, and how we operate on strings. In C++ we have a `string` type which makes our life *a lot* easier. Underneath this `string` type there is still a `char*` - we just don't need to interact with it directly. The `string` type is provided in the `string` header in C++ (note again no `.h`).

As an example application, let us rebuild our command line example (similar to Listing 1.17) in C++. The code is below:

```
#include <iostream>
#include <string>

int main(int argc, char **argv)
```

```
 5 {
 6     // Declare an empty string
 7     std::string command = "";
 8     // Loop round command lines, appending to the string
 9     for (int i = 0; i < argc; ++i)
10         command += std::string(argv[i]) + std::string(" ");
11 
12     // Print the command line
13     std::cout << command << std::endl;
14     // Print out the length of the string
15     std::cout << "Command line is " << command.size() << "
           characters long" << std::endl;
16 
17     return 0;
18 }
```

Listing 5.3: Using the C++ string Type

Here we are using the `std::string` object. As with `cout` and `cin`, `string` is part of the *Standard Library* namespace (hence the `std::` part).

Let us look at some of these lines in more detail:

line 2 - here we have included the `string` header file

line 7 - we declare an initial `string` with no text. Notice we can initialise a C++ `string` as we did with strings in C using ". Consider this a string initializer for C based languages.

line 10 - here we join the `string` objects together. Notice that we can just use the + operator to do this. The + operator concatenates `string` objects together. Line 10 effectively says *Add to command (the += part) the next command line argument and a space*

line 13 - here we print the command line string using `cout`. Note that `cout` will happily stream a `string` object for us.

line 15 - here we print the length of the `string` object. Note that we can get that by calling the `size` method on the `string` object. The `string` maintains its length for us so we can call the `size` method to access the value.

line 15 - also note here that we are streaming multiple values to `cout`. This allows us to create more complicated outputs if we need to.

Let us consider that you have compiled this application into something called `cpp_strings`. Then if you run the application using the command `cpp_strings hello world` you will get following output:

```
cpp_strings hello world
Command line is 24 characters long
```

Listing 5.4: Output from C++ string Application

5.3.1 Exercise

Write the equivalent C++ application for reading in a name and saying hello to the person - something similar to Listing 1.9. You will need to use the stream input operator and a `string` object. The other object you need is `std::cin` (*Console*

INput). To give you some guidance in how to structure the statement use the following:

From console input read into name (where *name* is the `string` you are reading into).

5.4 Accessing Raw String from C++ `string`

We can rewrite the above application by accessing the raw C string and then calling `printf` (therefore mixing C and C++). This code is shown below:

```
 1 #include <cstdio>
 2 #include <string>
 3 
 4 // A function that will print a C string
 5 void print(const char *str)
 6 {
 7     printf(str);
 8 }
 9 
10 int main(int argc, char **argv)
11 {
12     // Declare an empty string
13     std::string command = "";
14     // Loop round command lines, appending to the string
15     for (int i = 0; i < argc; ++i)
16     {
17         command += std::string(argv[i]);
18       command += std::string(" ");
19     }
20 
21     // Get the C string (raw string) from the C++ string
22     const char *str = command.c_str();
23 
24     // Call print
25     print(str);
26 
27     return 0;
28 }
```

Listing 5.5: Using `c_str()` to Access the Raw String

We are combining a few different ideas here. Let us look at these in turn:

line 1 - here we are including the `cstdio` header (again no `.h`). This header provides C++ with access to the functionality provided in the C `stdio.h` header. We will return to including C libraries in C++ at the end of this unit.

line 5 - we have declared a function that will print the value. Notice the use of `const` here. We are stating that the value in the `char*` will not change. We will be looking at `const` in more detail later in the unit.

line 7 - we are calling the standard `printf` function from C in our C++ code. As stated previously this is perfectly acceptable.

line 22 - this is where we are accessing the raw C string from our C++ `string` object. This is done using the `c_str` method on `string`. Note the return type - it is a `const char*`. This is because you should not be modifying this value outside of the string object - hence it is a const. The value you have (the pointer to a location in memory) points to the exact same location as the string object is storing its char*.

In general we don't have to write code that accesses the raw C string of a C++ string. However, you might find that sometimes you are working with a predominantly C based library or framework (for example OpenGL, OpenCL, CUDA, etc.) and you want to have the simplicity of C++ string objects but have to give the relevant function calls char*. The access to c_str provides this.

The output from this application should be the same as the previous application.

5.5 Reading Input from the Command Line with cin

You should have attempted this in an exercise already, so hopefully nothing here is that new. Just in case though, let us see how we capture input from the command line in C++. The following program emulates that of Listing 1.9 for C but in C++.

```cpp
#include <iostream>
#include <string>

int main(int argc, char **argv)
{
    // Declare strings to store name
    std::string first_name;
    std::string last_name;
    std::string full_name;

    // Prompt for first name
    std::cout << "Please enter your first name: ";
    // Read in first_name using cin
    std::cin >> first_name;

    // Check if your name is Kevin
    if (first_name == "Kevin")
        std::cout << "Hey! Another Kevin\n";
    else
        std::cout << "Oh well\n";

    // Output number of characters entered
    std::cout << "You entered " << first_name.length() << "
       characters" << std::endl;
    // Prompt for last name
    std::cout << "Please enter your last name: ";
    std::cin >> last_name;
    // Output number of characters entered
    std::cout << "You entered " << last_name.length() << "
       characters" << std::endl;

    // Join the strings
    full_name = first_name + std::string(" ") + last_name;

    // Print name
    std::cout << "Your full name is " << full_name << " which is ";
    std::cout << full_name.length() << " characters long" << std::
       endl;;

37     return 0;
38 }
```

Listing 5.6: Using `cin` to Read From the Command Line

Let us look at some of these lines, particularly where we have done things differently:

lines 7 to 9 - declare the C++ `string` objects we are going to use

line 14 - we read in from `cin` the `first_name` value using the stream input (`>>`) operator

line 17 - we perform a string comparison. Note now we can just use the standard equality operator (`==`) rather than calling a function to compare. The equality operator will perform the same process.

line 23 - we use the `length` method to get the length of the string. This will return the same value as `size` and therefore is interchangeable.

As you can see, working with input-output and strings is easier in C++ than C. This is one of the benefits of having object-orientation. We can operate on our data more effectively and therefore reduce our code.

5.6 Using getline to Read Lines of Text

OK, this is our final example of working with the C++ `string` object before moving onto the main point of this unit. Here we are going to use a nice helper function provided in the C++ Standard Library known as `getline`. This function allows us to read an entire line from an input stream until we encounter the newline character. This allows us to work around the problem of having space characters in our input streams. The example application is below. It simply prompts for and reads in someone's full name.

```
#include <iostream>
#include <string>

int main(int argc, char **argv)
{
    // Declare string to store name
    std::string full_name;

    // Prompt for name
    std::cout << "Please enter your full name: ";
    // Read in full_name using cin and getline
    std::getline(std::cin, full_name);

    // Print name
    std::cout << "Your full name is " << full_name << " which is ";
    std::cout << full_name.length() << " characters long" << std::endl;;

    return 0;
}
```

Listing 5.7: Using `getline`

Line 12 is where we are using the `getline` function. Notice is takes two parameters:

1. the stream to read from. This could be the command line, a file, or something similar

2. the `string` to read into

Compile and run this application to get an idea of the result. You should understand everything else about this code by now.

5.7 Passing by Value (Copying Data)

With that short tour of C++ focusing on command line interaction and strings we are ready to move onto the main focus of this unit - how values are passed to functions / operations in C and 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 reference to the function / operation (*pass-by-reference*)

3. Passing a value by providing a pointer to the function / operation (*pass-by-pointer*)

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.

```cpp
#include <iostream>

void foo(int x)
{
    std::cout << "Start of function, x = " << x << std::endl;
    x = 20;
    std::cout << "End of function, x = " << x << std::endl;
}

int main(int argc, char **argv)
{
    int x = 10;
    std::cout << "Before function call, x = " << x << std::endl;
    foo(x);
    std::cout << "After function call, x = " << x << std::endl;

    return 0;
}
```

Listing 5.8: 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 5.1.

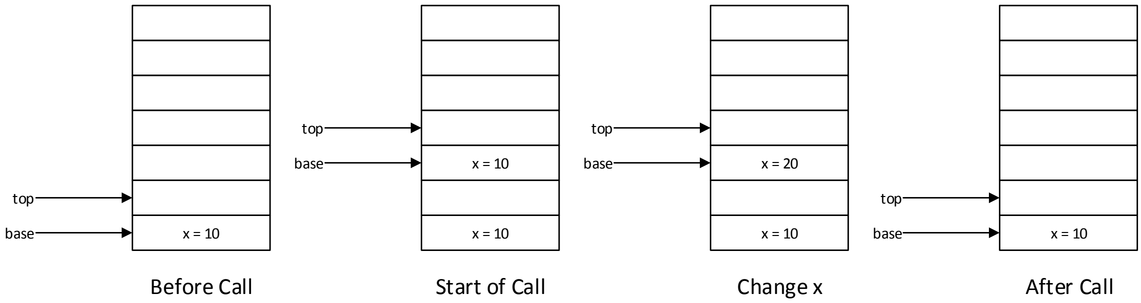


Figure 5.1: Interpretation of the Stack During Pass-by-Value

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 5.9: 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.

5.8 Passing by Reference

Now let us move onto working with *pass-by-reference*. This technique allows us to avoid passing a copy of a value to a function / operation but rather pass a *reference* to the value. This means that the function / operation is directly accessing the same value as the caller of the function.

When it comes to calling a function / operation that uses pass-by-reference, you as the caller of the function / operation has to do nothing different. It is the function / operation that declares that is taking in a reference to a value. It does this by declaring a parameter as a reference using the *reference-type* specifier. This is the ampersand (`&`) character. For example, to pass a parameter using pass-by-reference we would declare our function / operation as follows:

```
return-type function-name(parameter-type &name) { ... }
```

Any type can have the *reference-type* specifier added to it. For example, a reference to a `char` is of type `char&`. An `int` is of type `int&`. And so on.

The & Specifier and the & Operator

We are now getting into one of the areas that can really confuse new C++ programmers - the use of the & symbol. We now have four different uses for this symbol. These are summarised in the following table.

Use	Example	Description
Bitwise and operator	`int c = a & c`	Performs a bitwise and operation on two values. This takes the bit pattern of the value and logically ands the individual bits.
Logical and operator	`bool c = a && b`	Performs a logical and on two boolean values.
Address of operator	`int *a = &b`	Gets the address (to create a pointer) of a value.
Reference to specifier	`int &a = b`	Declares a variable as a reference type.

The main confusion comes between the use of & as an operator to get a pointer to a value and declaring a variable as a reference type. This is because they are related. Both of these capabilities rely on us not storing a value directly but having a way of accessing the value.

To help clarify the difference, try and remember the following:

- A pointer type is declared using * (e.g. `char*`). To get a pointer to a value, we use the & operator on a variable. When using & on a variable, we are getting a pointer to it.

- A reference type is declared using & (e.g. `char&`). To get a reference to a value, we simply assign the variable to the reference type. We use & to declare types (e.g. variable declarations, parameter declarations) as we do with * for pointer types.

References may seem strange but you have actually been using them all the time in modern object-oriented languages. Everything in Java and C# (except the primitive types such `int`) are passed as references. This means that in all likelihood you already think in references. This can make working in *pass-by-value* a bigger headache. However, as you may notice, we can pass any type as a reference in C++ (using the & specifier). In Java we cannot, although C# does provide some of this capability.

With that introduction to pass-by-reference let us move onto our example application. This is the same as the previous pass-by-value one but this time we use a reference for the parameter. The code is below.

```
#include <iostream>

void foo(int &x)
```

```
 4 {
 5     std::cout << "Start of function, x = " << x << std::endl;
 6     x = 20;
 7     std::cout << "End of function, x = " << x << std::endl;
 8 }
 9
10 int main(int argc, char **argv)
11 {
12     int x = 10;
13     std::cout << "Before function call, x = " << x << std::endl;
14     foo(x);
15     std::cout << "After function call, x = " << x << std::endl;
16
17     return 0;
18 }
```

Listing 5.10: Passing a Parameter by Reference

Our change is on line 3 where we declare the parameter `x`. Now the type is `int&` rather than just `int`. This means that we are modifying the exact same value that `main` has in `foo`. Figure 5.2 provides an example.

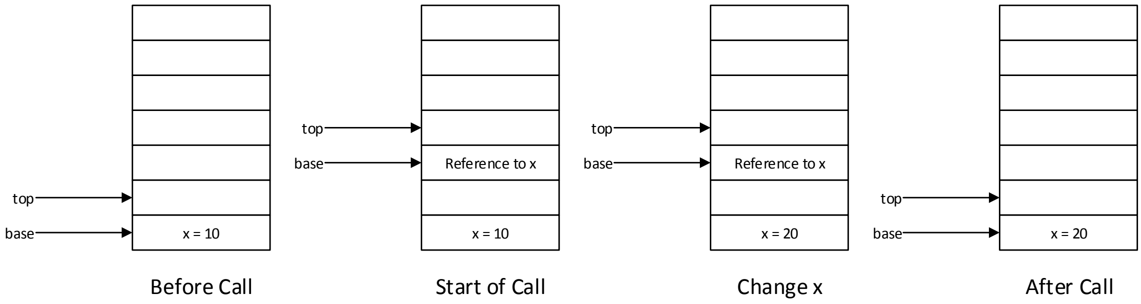


Figure 5.2: Interpretation of the Stack Using Pass-by-Reference

Running this application will provide the following output:

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

Listing 5.11: output from Pass-by-Reference Application

As we can see, the modification to `x` in the function is reflected in the main application. The reference means that we have not created a copy of the value.

Why use References?

There are a number of reasons for and against using references over copying data between functions. A few to consider are:

1. *Pass-by-value* copies data when the variable is passed. On a sized data types this is unlikely to concern us as the memory overhead and cost of copying is low. However, on anything larger than the size of copy for a reference we start to degrade performance. A reference is only 4 bytes in size (or 8 on a 64-bit operating system) and therefore can be copied in a single instruction. Anything larger than this on pass-by-value will be slower.

2. *Pass-by-Reference* allows you to modify (and therefore return) multiple values in a function. This has its advantages when trying to write some operations that need to output multiple values.

3. *Pass-by-Reference* does mean that you can modify values in a function. This could cause unforeseen behaviour if you are calling someone else's code.

4. *Pass-by-Reference* can lead to multiple objects having access to the same value. This can enable multiple objects to *mutate the state* (in other words modify the attributes) of the object and be difficult to track in large applications.

The developers of Java decided that pass-by-reference was the best method (they have a garbage collector to clean up memory - C++ doesn't which we will explore next unit). In practice, pass-by-reference is also recommended in C++, but it has it's limitations. For example, there is no such thing as a `null` reference in C++. This means a reference must also point to an initialised value. Also, a reference cannot change to point to a different location in memory. It is fixed and will also point to the same location. This has it's pros and cons. The pointer type is more versatile, but therefore also more volatile.

5.9 Example - Copying and Sorting by Value and Reference

Let us look at how pass-by-value impacts memory usage in comparison to pass-by-reference. We will build an application that performs a bubble sort on two large arrays of data in two different ways. The first way will involve pass-by-copy which will require two copies of the array to be in existence, and hence more memory being used. The second will involve pass-by-reference and hence reduce memory usage. To do this we will also use a `vector` to store our data.

What is a `vector` in C++

A `vector` in C++ terms is very similar to a standard array except it provides mechanisms to add, remove, insert, etc. The `vector` will resize itself based on requirements. This means that if you add a value to the `vector` it will increase in size, and if you remove an item the `vector` will decrease in size.

It is very likely that you have used a data store object that resizes in other languages. In Java this is known as an `ArrayList`. We will use `vector` in places from now on, but we will cover data structures in more depth in a later unit.

A `vector` has a type associated with it. This is declared in angle brackets in the variable declaration. For example, to declare a `vector` that stores type `int` we use `vector<int>`. This uses a technique called *templates* which allow us to provide compile time type information. We will only briefly cover templates as they can be used for far more sophisticated purposes than we can fully cover in the module.

Our test application is below.

```
#include <iostream>
#include <vector>

void sort_copy(std::vector<int> data)
{
    // Iterate through each value
    for (int i = 0; i < data.size(); ++i)
    {
        // Loop through values above index i
        for (int j = 0; j < data.size() - (i + 1); ++j)
        {
            // Test if data[j] > data[j + 1]
            if (data[j] > data[j + 1])
            {
                // Swap values
                int temp = data[j + 1];
                data[j + 1] = data[j];
                data[j] = temp;
            }
        }
        if (i % 1000 == 0)
      std::cout << ((float)i / (float)data.size()) * 100.0f << "% sorted" << std::endl;
    }
}

void sort_reference(std::vector<int> &data)
{
    // Iterate through each value
    for (int i = 0; i < data.size(); ++i)
    {
        // Loop through values above index i
        for (int j = 0; j < data.size() - (i + 1); ++j)
        {
            // Test if data[j] > data[j + 1]
            if (data[j] > data[j + 1])
            {
                // Swap values
                int temp = data[j + 1];
                data[j + 1] = data[j];
                data[j] = temp;
            }
        }
        if (i % 1000 == 0)
            std::cout << ((float)i / (float)data.size()) * 100.0f << "% sorted" << std::endl;
    }
}

int main(int argc, char **argv)
{
    std::vector<int> data;
    for (int i = 0; i < 262144; ++i)
        data.push_back(262144 - i);

    std::cout << "Sorting by copy..." << std::endl;
    sort_copy(data);
    std::cout << "Sorting by reference..." << std::endl;
    sort_reference(data);

59      return 0;
60  }
```

Listing 5.12: Passing a Vector by Value and Reference

Let us point out a few lines of interest:

line 4 - here we are passing the vector by *value*. Note that we don't use any specifier on the data type for the parameter.

lines 7, etc. - we use the method `size` on the `vector`. This provides the number of elements currently stored in the `vector`.

line 13, etc. - notice that we can use standard array notation to access members of the `vector`, e.g. `data[i]`. Within the `vector` is an actual array used to store the data. C++ provides *operator overloading* which allows us to access this array.

line 26 - here we are passing the vector by *reference*. We have used the `&` specifier on the data type for the parameter.

line 52 - we use the `push_back` method on the `vector` to add values to the end of the data store. For the first call to `push_back` we set the 0^{th} element, for the second call the 1^st element, and so on.

line 55 - we call the pass-by-value version of sort. Notice we don't have to use any additional code to call with pass-by-value.

line 57 - we call the pass-by-reference version of sort. Notice we don't have to use any additional code to call with pass-by-reference.

The size of the `vector` data in memory is approximately 1 megabyte. When we use the pass-by-value version, we create a copy of the `vector` and therefore add another 1 megabyte of memory usage. This can be seen using the Windows Task Manager.

5.9.1 Exercise

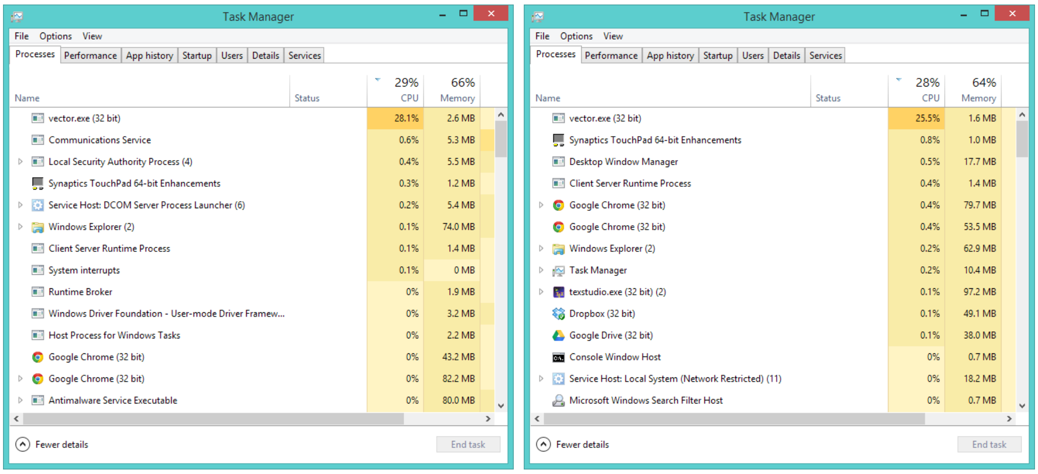
To truly test this application you will need to open Windows Task Manager. Watch the application memory usage when the application has copied the vector, and also when it has passed by reference. An example is shown in Figure 5.3.

5.10 Array Library in C++

We have now introduced enough C++ to allow us to reimplement our simple array library using C++ objects and techniques. Remember that we created three pieces of functionality:

1. Searching for an item in the array
2. Sorting data in the array
3. Generating random values to fill the array

We will not implement the search method this time - mainly because the `vector` provides this capability. To be fair, C++ provides a sort capability which you can look at in the exercises. The other change we will make is the use of the C++ `vector` rather than a raw array. Let us look at the two other functions in turn.


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task
Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task

Name	Status	29% CPU	66% Memory
vector.exe (32 bit)		28.1%	2.6 MB
Communications Service		0.6%	5.3 MB
Local Security Authority Process (4)		0.4%	5.5 MB
Synaptics TouchPad 64-bit Enhancements		0.3%	1.2 MB
Service Host: DCOM Server Process Launcher (6)		0.2%	5.4 MB
Windows Explorer (2)		0.1%	74.0 MB
Client Server Runtime Process		0.1%	1.4 MB
System interrupts		0.1%	0 MB
Runtime Broker		0%	1.9 MB
Windows Driver Foundation - User-mode Driver Framew...		0%	3.2 MB
Host Process for Windows Tasks		0%	2.2 MB
Google Chrome (32 bit)		0%	43.2 MB
Google Chrome (32 bit)		0%	82.2 MB
Antimalware Service Executable		0%	80.0 MB

Name	Status	28% CPU	64% Memory
vector.exe (32 bit)		25.5%	1.6 MB
Synaptics TouchPad 64-bit Enhancements		0.8%	1.0 MB
Desktop Window Manager		0.5%	17.7 MB
Client Server Runtime Process		0.4%	1.4 MB
Google Chrome (32 bit)		0.4%	79.7 MB
Google Chrome (32 bit)		0.4%	53.5 MB
Windows Explorer (2)		0.2%	62.9 MB
Task Manager		0.2%	10.4 MB
texstudio.exe (32 bit) (2)		0.1%	97.2 MB
Dropbox (32 bit)		0.1%	49.1 MB
Google Drive (32 bit)		0.1%	38.0 MB
Console Window Host		0%	0.7 MB
Service Host: Local System (Network Restricted) (11)		0%	18.2 MB
Microsoft Windows Search Filter Host		0%	0.7 MB

Figure 5.3: Task Manager During Pass-by-Copy (left) and Pass-by-Reference (right)

5.10.1 Sorting

Our C++ header for sort is again called `sort.h`. Remember that it just takes the array data and its size. Now we are using a `vector` the size is provided as a method. Our `sort.h` is declared below.

```cpp
#pragma once

#include <vector>

void sort(std::vector<int> &data);
```

Listing 5.13: `sort.h` Header File in C++

The implementation undertakes the same bubble sort approach as we did in the C version. The only difference is that we are using the `vector` data type rather than the raw array. The code for `sort.cpp` is below.

```cpp
#include "sort.h"
#include <iostream>

void sort(std::vector<int> &data)
{
    // Iterate through each value
    for (int i = 0; i < data.size(); ++i)
    {
        // Loop through values above index i
        for (int j = 0; j < data.size() - (i + 1); ++j)
        {
            // Test if data[j] > data[j + 1]
            if (data[j] > data[j + 1])
            {
                // Swap values
                int temp = data[j + 1];
                data[j + 1] = data[j];
                data[j] = temp;
            }
        }
        if (i % 1000 == 0)
            std::cout << ((float)i / (float)data.size()) * 100.0f
                << "% sorted" << std::endl;
    }
}
```

Listing 5.14: `sort.cpp` Code File in C++

5.10.2 Generating

Our C generation method required our data to generate into and its size. For the C++ version we will just take a size value and return a new `vector` of that size filled with random values. As such, our header file `generate.h` is defined as follows:

```cpp
#pragma once

#include <vector>

std::vector<int> generate(int size);
```

Listing 5.15: `generate.h` Header File in C++

The implementation will use C++ random mechanisms to create the `vector`. The code is below. We will look at C++ random after the code is introduced.

```cpp
#include "generate.h"
#include <random>

std::vector<int> generate(int size)
{
    // Create random generator
    std::random_device rd;
    // Create distribution
    std::uniform_int_distribution<int> dist;
    // Generate random numbers
    std::vector<int> data;
    for (int i = 0; i < size; ++i)
        data.push_back(dist(rd));

    return data;
}
```

Listing 5.16: `generate.cpp` Code File in C++

Random Numbers in C++

Random number generation is a relatively new feature in C++. Until the C++11 standard (released in 2011), C++ programmers used the same random function as C. With the introduction of C++11, C++ programmers have quite a wide ranging set of objects to generate random numbers.

C++ random number generation uses two objects:

random number generation engine - this object is responsible for generating random numbers using a particular algorithm. This is the `random_device` object in our code.

value distribution - this object can take a random engine and return a value of the required type and within a given range. This is our `uniform_int_distribution` object in our code. It allows us to get values of type `int` from the random engine. We could also define a minimum and maximum value to be produced by providing these as parameters.

We can get random numbers without using a distribution, but generally you will want to use one. To get a value within the distribution we use the following code:

```
type val = distribution(engine);
```

5.10.3 Test Application

We can now build a test application to test our library. This is shown below. As can be seen, using C++ and object orientation reduces our code requirement.

```cpp
#include "generate.h"
#include "sort.h"
#include <iostream>

int main(int argc, char **argv)
{
    std::vector<int> data = generate(65536);
    sort(data);

    // Print the first 100 values
    for (int i = 0; i < 100; ++i)
        std::cout << data[i] << std::endl;

    return 0;
}
```

Listing 5.17: Test Application for C++ Array Library

5.10.4 Exercise

You can now build the application (remember - you should be using make files for this). There is a library component (the sort and generate parts), and the main application. Ensure that you are building a separate library that you link in the main application build. We have already covered this, so go back and review the content to ensure you can replicate it. An example output is given below:

```
... previous lines

2416421
2453633
2472376
2515740
2585328
2588164
2609197
2623708
2625554
2660361
2669049
2669493
2678787
2750678
2751473
2772010
```

Listing 5.18: Output from C++ Array Library Test Application

5.11 Reading Files in C++

In the last unit we took our array library and combined it with some file input-output. We will repeat this now for C++ and see the difference working with object-orientation provides. C++ works on the basis of *streams*, which we have already seen with `cout` and `cin`. Files, once opened, are also just streams we can read from and write to. Table 5.1 describes some of the common base stream types in C++.

Type	**Friendly Name**	**Description**
`istream`	input stream	A stream that data can be read from.
`ostream`	output stream	A stream that data can be written to.
`iostream`	input-output stream	A stream that can be read from and written to
`fstream`	file stream	Represents an input-output stream that points to a file. `ifstream` and `ofstream` also exist.
`stringstream`	string stream	Provides a stream that can be written to to create a `string`. Can be used instead of appending.

Table 5.1: Common Stream Types in C++

The one we are interested in here is `fstream`, or more specifically `ifstream` to read data from. We will look at `ofstream` in the next section.

Working with `ifstream`

Before looking at our application let us introduce how we work with the `ifstream` type.

Creating a `ifstream` - to create a `ifstream` object (and either open or create a file accordingly) we use the call `ifstream` *`var-name`*`(`*`filename, mode`*`);`. The *filename* parameter is a `string` representing the filename. The mode determines the mode we open the file in (e.g. binary).

Seeking in a File - we can seek to a particular position in a file using the `seekg` command. We can use this to move to the end of a file and then get the position using `tellg`.

Getting the Current File Position - we can use `tellg` to tell us which position in the file (byte number) we are currently accessing.

Reading in Data - we can use the streaming operator to read in individual values. However, we can also use `read` to read in a set amount of data into an area of memory

Closing a `ifstream` - the method `close` is used to close a `ifstream`

Our application to read from a data file is below. With the `ifstream` operations described above you should be able to understand what we are doing.

```
 1 #include <iostream>
 2 #include <fstream>
 3 #include <string>
 4 #include "sort.h"
 5
 6 std::vector<int> readfile(std::string &filename)
 7 {
 8     // Open the file
 9     std::ifstream file(filename, std::ios::binary);
10     // Get the size of the file - seek to end
11     file.seekg(0, file.end);
12     std::streampos size = file.tellg();
13     // Seek back to start
14     file.seekg(0, file.beg);
15     // Number of elements is size / sizeof(int)
16     int elements = size / sizeof(int);
17     // Create an array of data to read into
18     int *temp = new int[elements];
19     // Read in data
20     file.read((char*)temp, size);
21     // Close the file
22     file.close();
23     // Copy data into the vector
24     std::vector<int> data(temp, temp + elements);
25     // Delete the data
26     delete[] temp;
27     // Return the vector
28     return data;
29 }
30
31 int main(int argc, char **argv)
32 {
33     // Read in a vector
34     std::vector<int> data = readfile(std::string("numbers.dat"));
35     // Print the vector size
36     std::cout << "Numbers read = " << data.size() << std::endl;
37     // Sort the vector
38     sort(data);
39     // Output first 100 numbers
40     for (int i = 0; i < 100; ++i)
41         std::cout << data[i] << std::endl;
42
43     return 0;
44 }
```

Listing 5.19: Reading Binary Files in C++

Let us look at some of these lines in more detail.

line 6 - note that we are passing the name of the file as a reference to `readFile`. This is because we don't need a copy of the filename in our `readFile` operation.

line 9 - here we open our file. Note the mode we open with - `std::ios::binary`. This means that we are opening the file as a binary file.

line 11 - we use `seekg` to seek to the end of the file (0 from `end` of `file`). This is important for the next line.

line 12 - we get the position in the file using `tellg`. We now know the size of the file in bytes.

line 14 - we seek back to the start of the file (0 from `beg`inning of `file`).

line 16 - we calculate the number of `int` values in the file by dividing its size by the size of an `int`.

line 18 - we allocate memory to store the file contents using the `new` operator. We will cover this in more detail in the next unit.

line 20 - we read the file into our allocated memory. Note that we have to cast to a `char*` for this to happen.

line 22 - we close the file.

line 24 - this is a new way of creating our `vector` and might seem a bit strange. What we are doing is giving `vector` a starting and ending memory location, which is used to copy the data into the `vector`. The starting memory location is the pointer representing our memory we read the file into. The end location is this memory location plus the number of elements (which will be implicitly multiplied by the size of the data type). This line takes a bit of getting used to, so take your time and ensure you understand it.

line 26 - we free the allocated memory from line 18. This will be covered in the next unit.

Using `new` and `delete` in C++

In this example we have used to new concepts - `new` and `delete`. These commands are used in C++ to allocate memory and free it after we have finished with it. We have looked at these ideas a little in C. You are very likely used to working with `new` in Java. However, in C++ things are different in that we need to ensure that we free up any memory created using `new`. This is unlike Java where the garbage collector does this for us. Working out when we need to allocate and free memory is an important skill in C++.

5.12 Writing Files in C++

Let us now look at writing files in C++. The code for our application is below. This should be straightforward by now.

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

std::vector<std::string> readfile(std::string &filename)
{
    // Open file - default is text
    std::ifstream file(filename);
    // String to read into
    std::string line;
    // Data to return
13     std::vector<std::string> data;
14     // Read until end of file
15     while (std::getline(file, line))
16         data.push_back(line);
17     // Close file
18     file.close();
19     // Return data
20     return data;
21 }
22
23 void writefile(std::string &filename, std::vector<std::string> &
       data)
24 {
25     // Open file - default is text
26     std::ofstream file(filename);
27     // Write each line into file starting at end of vector
28     for (int i = data.size() - 1; i >= 0; --i)
29         file << data[i] << std::endl;
30     // Close the file
31     file.close();
32 }
33
34 int main(int argc, char **argv)
35 {
36     // Read in file
37     std::vector<std::string> data = readfile(std::string("sorted.
           txt"));
38     // Print lines read
39     std::cout << "Lines read = " << data.size() << std::endl;
40     writefile(std::string("reversed.txt"), data);
41
42     return 0;
43 }
```

Listing 5.20: Writing Text Files in C++

Line 15 uses the `get_line` command to read a line of the text file into a string. This call returns `true` as long as there are lines of code to read. This is why we use the `while` loop to fill the `vector` with read strings. The other line to take note of is line 29 where we using the streaming operator to write lines to the file.

5.13 `const` References

Our final look at references considers the use of `const`. Remember in C we used `const` to define constant values, whether it be variables or parameters. We can also declare references as `const`. This means that the value stored in the reference cannot be changed. Let us look at an example.

```
#include <iostream>
#include <string>

std::string join(const std::string &a, const std::string &b)
{
    // This line won't compile
    return a.append(b);
}

int main(int argc, char **argv)
{
```

```
12      std::string greeting = join(std::string("Hello,"), std::string(
            " World!"));
13
14      std::cout << greeting << std::endl;
15
16      return 0;
17  }
```

Listing 5.21: Passing `const` References

The `append` method used on line 6 is not `const`, and therefore your code won't compile. However, if we modify our `join` function to use the append operator (`+`) we create a copy of the strings and therefore get around the problem. This code is below.

```
std::string join(const std::string &a, const std::string &b)
{
    // This line will compile
    return a + b;
}
```

Listing 5.22: Adding Two `const string` Values Together in C++

You might ask then why do we pass by `const` at all. For one reason, safety - we are stating we are not going to modify the value. For another reason, the compiler can actually improve the application performance if it knows a value is `const`. This can be an important benefit when writing optimised code, and you should consider using `const` whenever it is suitable.

5.13.1 Exercise

Can you write another version of `join` that uses the `append` method of `string`? This is possible. You must use the same function prototype as already defined (i.e. the values must be `const`).

5.14 Pointers

So references where the easier part of this unit. The more difficult part is understanding 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, except that we can make a pointer point to different parts of memory as required. Remember that a reference is fixed.

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 5.2.

Let us now implement our *pass-by-pointer* application as we did for pass-by-value and pass-by-reference. 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.

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 5.2: Pointer Operators and Specifiers

```cpp
#include <iostream>

void foo(int *x)
{
    std::cout << "Address of x in function = " << x << std::endl;
    std::cout << "Start of function, x = " << *x << std::endl;
    // Have to dereference pointer to change value
    *x = 20;
    std::cout << "End of function, x = " << *x << std::endl;
    std::cout << "Address of x at end of function = " << x << std::endl;
}

int main(int argc, char **argv)
{
    int x = 10;
    std::cout << "Starting address of x = " << &x << std::endl;
    std::cout << "Before function call, x = " << x << std::endl;
    // Have to pass the pointer (or address of) x to the function
    foo(&x);
    std::cout << "After function call, x = " << x << std::endl;
    std::cout << "End address of x = " << &x << std::endl;

    return 0;
}
```

Listing 5.23: Passing a Value as a Pointer

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

```
Starting address of x = 003FFC30
Before function call, x = 10
Address of x in function = 003FFC30
Start of function, x = 10
End of function, x = 20
Address of x at end of function = 003FFC30
After function call, x = 20
End address of x = 003FFC30
```

Listing 5.24: Output from Pass-by-Pointer Application

5.15 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 and 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 / 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 C++ 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++ we can use the `new` operator to create an array of the required size at runtime:

type `*name = new` *type*`[size];`

This is how you are likely to remember array allocation in Java (apart from the pointer part).

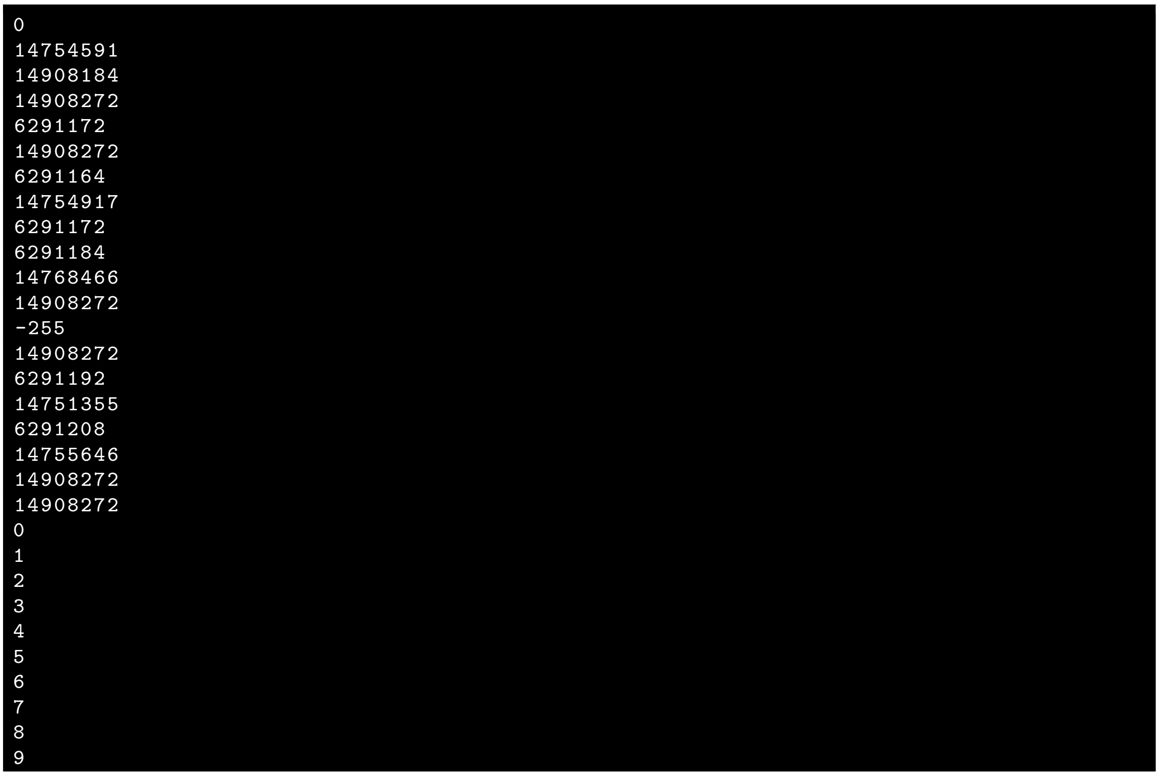
In C and C++ arrays are a pointer to the start of the memory block where the memory is located. The size of the array is not stored however (unlike in Java) so we have to pass this value around. This is why working with `vector` in C++ is almost always a better choice.

An example array application is below:

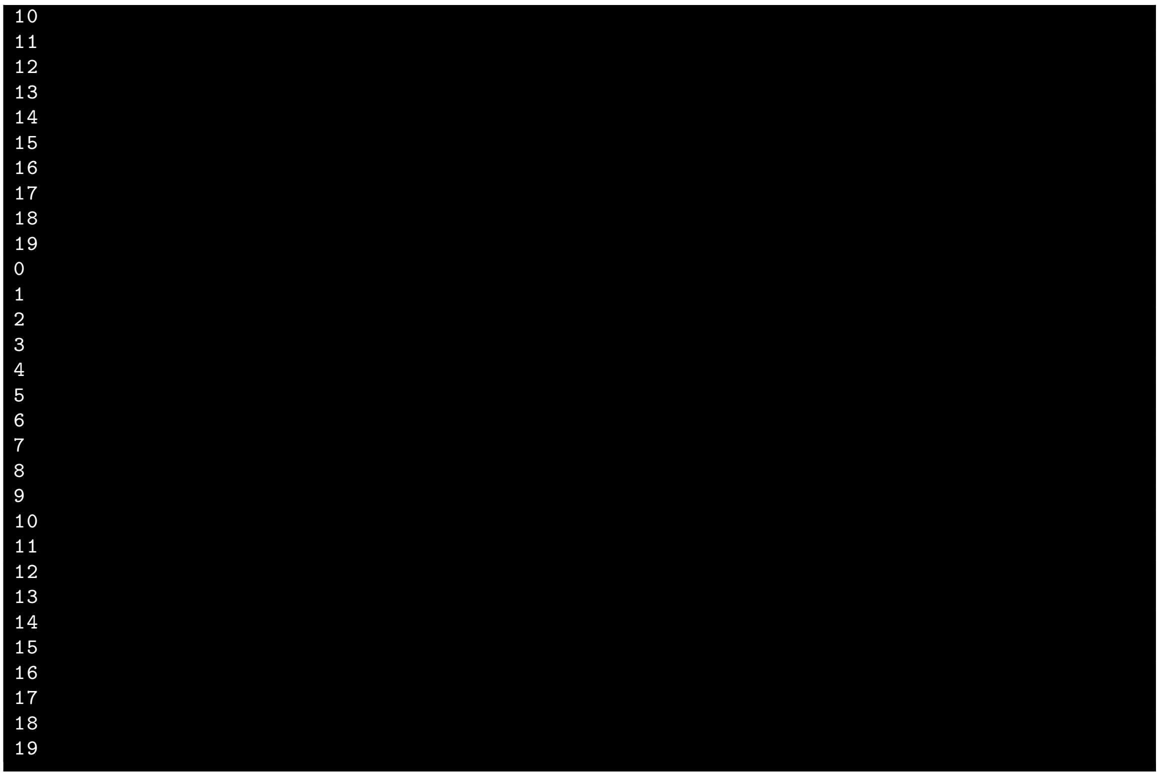
```cpp
#include <iostream>

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 = new 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)
        std::cout << data[i] << std::endl;
35
36     data = create_array_new();
37     // Print out all elements
38     for (int i = 0; i < 20; ++i)
39         std::cout << data[i] << std::endl;
40     // Free the memory
41     delete[] data;
42     // Set to nullptr
43     data = nullptr;
44
45     // Create array from pointer
46     // This will cause a memory allocation error
47     // nullptr (memory address 0) cannot be allocated to
48     create_array(data);
49
50     // Allocate memory
51     data = new int[20];
52     // Create array from pointer
53     create_array(data);
54     // Print out all elements
55     for (int i = 0; i < 20; ++i)
56         std::cout << data[i] << std::endl;
57     // Free the memory
58     delete[] data;
59     // Set to nullptr
60     data = nullptr;
61
62     return 0;
63 }
```

Listing 5.25: Passing Arrays as Pointers

Compiling this version of the code will cause a runtime error 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 5.26: 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 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 5.4 provides an illustration of what is happening.

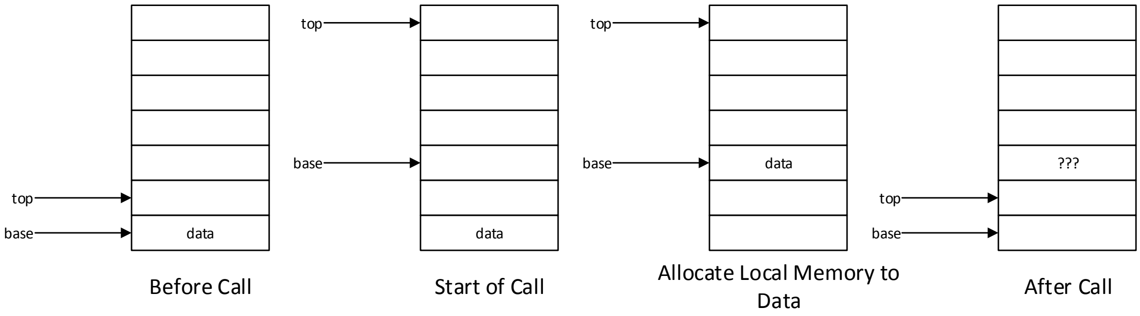


Figure 5.4: Data and Stack Corruption Example - Returning Non-Valid Memory

5.16 `const` Pointers and Pointers to `const`

We covered `const` in relation to references previously. 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 5.3 illustrates the different approaches.

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

`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 5.3: `const`ness of Pointers

the commented out lines. This is so you can see the compiler error that C++ will provide.

```cpp
#include <iostream>

void foo(const int *x)
{
    std::cout << "Address of x in function = " << x << std::endl;
    std::cout << "Start of function, x = " << *x << std::endl;
    // Wont compile - value pointed to is const
    *x = 20;
    std::cout << "End of function, x = " << *x << std::endl;
    std::cout << "Address of x at end of function = " << x << std::
        endl;
}

void foo2(int *const x)
{
    std::cout << "Address of x in function = " << x << std::endl;
    std::cout << "Start of function, x = " << *x << std::endl;
    // Will compile - pointer is const, not value pointed to
    *x = 20;
    std::cout << "End of function, x = " << *x << std::endl;
    std::cout << "Address of x at end of function = " << x << std::
        endl;
}

void foo3(int *const x)
{
    std::cout << "Address of x in function = " << x << std::endl;
    std::cout << "Start of function, x = " << *x << std::endl;
    // Won't compile - trying to change address pointed to
    x = nullptr;
    std::cout << "End of function, x = " << *x << std::endl;
    std::cout << "Address of x at end of function = " << x << std::
        endl;
}

int main(int argc, char **argv)
{
    int x = 10;
    std::cout << "Starting address of x = " << &x << std::endl;
    std::cout << "Before function call, x = " << x << std::endl;
    // Have to pass the pointer (or address of) x to the function
    foo2(&x);
    std::cout << "After function call, x = " << x << std::endl;
    std::cout << "End address of x = " << &x << std::endl;

    return 0;
}
```

Listing 5.27: `const` Pointers and Pointers to `const`

The compiler error provided by Microsoft's C++ compiler is as follows:

```
        cl const_pointer.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.30501 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

const_pointer.cpp
C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\INCLUDE\xlocale(337) : wa
rning C4530: C++ exception handler used, but unwind semantics are not enabled. S
pecify /EHsc
const_pointer.cpp(8) : error C3892: 'x' : you cannot assign to a variable that i
s const
const_pointer.cpp(28) : error C3892: 'x' : you cannot assign to a variable that
is const
```

Listing 5.28: Compiler Error from Trying to Modify a `const`

What is `nullptr`?

We have introduced another new term in this code - `nullptr`. This is a new addition to C++11 and provides the equivalent of `null` in Java. `nullptr` 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.

5.17 Namespaces

One final idea from C++ we will introduce in this unit is the idea of *namespaces*. A namespace is just a nice way of managing a collection of code that we consider to be part of a unit (such as a library). We've actually been using namespaces since the start of our C++ work. This is the `std` part we have been putting in front of our objects from the standard library.

There is a debate amongst C++ programmers about when you should use the full object name, but usually you want to reduce the amount of typing you have to undertake when coding. Typically in our main application we want to just tell the compiler that we are using a particular namespace and then just use the object names directly. We do this using the `using` statement:

```
using namespace std;
```

Because we have told C++ that we are using a particular namespace we can now access our objects directly. So instead of writing `std::string` we can just write `string`. *From this point onwards our applications will use the `std` namespace.* This will always occur in our main applications. However, in header files we will not use the namespace. This is common practice in C++ code.

As an example application, let us rewrite our file I/O code with a `using` statement.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

vector<string> readfile(string &filename)
{
    // Open file - default is text
    ifstream file(filename);
    // String to read into
    string line;
    // Data to return
    vector<std::string> data;
    // Read until end of file
    while (getline(file, line))
        data.push_back(line);
    // Close file
    file.close();
    // Return data
    return data;
}

void writefile(string &filename, vector<string> &data)
{
    // Open file - default is text
    ofstream file(filename);
    // Write each line into file starting at end of vector
    for (int i = data.size() - 1; i >= 0; --i)
        file << data[i] << endl;
    // Close the file
    file.close();
}

int main(int argc, char **argv)
{
    // Read in file
    vector<string> data = readfile(string("sorted.txt"));
    // Print lines read
    cout << "Lines read = " << data.size() << endl;
    writefile(string("reversed.txt"), data);

    return 0;
}
```

Listing 5.29: Using Namespaces in C++

As can be seen, our code is a bit simpler and there is less to write. You are likely to come across different approaches to using namespaces in other peoples C++ code, but the approach taken in this module is that described above. This is the expected coding practice of the module.

5.18 Using C Libraries in C++

As a final example, let us look at how we use the C standard libraries in our C++ code. All the C libraries have been provided in C++, declared in relevant header files. These files have the same name as their C equivalents with the following two differences:

1. The name of the header file has `c` at the start of it
2. The header file has no `.h` part

So for example the header file `stdio.h` becomes `cstdio`. Let us build an example application using the C libraries in C++. This application is just our data sizes one we built earlier.

```
#include <cstdlib>
#include <cstdio>
#include <climits>
#include <cfloat>

int main(int argc, char **argv)
{
    printf("unsigned char is %d bytes, min value %u, max value %u\n", sizeof(unsigned char), 0, UCHAR_MAX);
    printf("signed char is %d bytes, min value %d, max value %d\n", sizeof(signed char), SCHAR_MIN, SCHAR_MAX);
    printf("char is %d bytes, min value %d, max value %d\n", sizeof(char), CHAR_MIN, CHAR_MAX);
    printf("unsigned short is %d bytes, min value %u, max value %d\n", sizeof(unsigned short), 0, USHRT_MAX);
    printf("short is %d bytes, min value %d, max value %d\n", sizeof(short), SHRT_MIN, SHRT_MAX);
    printf("unsigned int is %d bytes, min value %u, max value %u\n", sizeof(unsigned int), 0, UINT_MAX);
    printf("int is %d bytes, min value %d, max value %d\n", sizeof(int), INT_MIN, INT_MAX);
    printf("unsigned long is %d bytes, min value %lu, max value %lu\n", sizeof(unsigned long), 0, ULONG_MAX);
    printf("long is %d bytes, min value %ld, max value %ld\n", sizeof(long), LONG_MIN, LONG_MAX);
    printf("unsigned long long is %d bytes, min value %llu, max value %llu\n", sizeof(unsigned long long), 0ULL, ULLONG_MAX);
    printf("long long is %d bytes, min value %lld, max value %lld\n", sizeof(long long), LLONG_MIN, LLONG_MAX);
    printf("float is %d bytes, min value %e, max value %e\n", sizeof(float), FLT_MIN, FLT_MAX);
    printf("double is %d bytes, min value %e, max value %e\n", sizeof(double), DBL_MIN, DBL_MAX);
    printf("long double is %d bytes, min value %e, max value %e\n", sizeof(long double), LDBL_MIN, LDBL_MAX);

    // Pointer sizes
    printf("char* is %d bytes\n", sizeof(char*));
    printf("short* is %d bytes\n", sizeof(short*));
    printf("int* is %d bytes\n", sizeof(int*));

    return 0;
}
```

Listing 5.30: Using C Libraries in C++

We will visit more C++ throughout the rest of the module. This is the language we shall be working in from now on unless stated otherwise. The only real C work we will do in future is looking at how C manages memory. We will introduce new C++ concepts as required.

5.19 Exercises

This has been a large unit, and you should familiarise yourself with the content. However, the following exercises will help you strengthen your understanding and knowledge of C++.

1. Investigate `std::setprecision` and how it can be used to determine the number of decimal places printed out on the command line. Write an application that tests the `std::setprecision` method on the command line to see the different results.

2. C++ provides `std::find` and `std::sort` methods. Investigate these and then use them to rewrite the array library developed in this unit using `std::vector`. In particular, compare the performance of the `sort` method with our trivial bubble sort.

3. Our `generate` function did seed its random (see discussion on `srand` previously). Discover how to seed a random engine in C++ and do this in your library to ensure the random numbers cannot be predicted.

4. Other C++ random engines exist. Investigate these random engines by using them in your array library. Look at the output results to understand how they generate values.

5. Modify the distribution in your array library so that numbers only up to 255 are generated.

6. **For the Brave** - we haven't covered certain aspects of call conventions in this unit. Investigate the `__cdecl`, `__stdcall` and `__fastcall` calls.

Unit 6

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/C++, working with memory is an important consideration. So far, our journey through C and 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 in C++ using references and pointers.

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. 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++.

6.1 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.

6.1.1 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 <iostream>

using namespace std;

int main(int argc, char **argv)
{
    // Outermost declaration
    int x = 10;
    cout << "Outermost x = " << x << endl;
    {
        // Now in new scope
        int x = 20;
        cout << "\tInner x = " << x << endl;
        {
            // Now even further in scope
            int x = 30;
            cout << "\t\tInner inner x = " << x << endl;
            {
                // Let's stop here
                int x = 40;
                cout << "\t\t\tInnermost x = " << x << endl;
                // Now unwind scope
            }
            cout << "\t\tInner inner x = " << x << endl;
        }
        cout << "\tInner x = " << x << endl;
    }
    cout << "Outermost x = " << x << endl;

    return 0;
}
```

Listing 6.1: 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 6.2: Output from Scope Test Application

6.1.2 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:

```
#include <iostream>

int main(int argc, char **argv)
{
    // Declare an int here
    int i = 10;
    {
        // Declare another int here - can access i
        int j = i * 2;
        {
            // Declare another int here - can access i and j
            int k = i + j;
13         }
14         // Compile error here - k not in scope
15         j = j + k;
16     }
17     // Compile error here - j not in scope
18     i = i + j;
19 
20     return 0;
21 }
```

Listing 6.3: 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:

```
D:\programming-fundamentals\code\unit-06>cl scope2.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.30501 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

scope2.cpp
C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\INCLUDE\xlocale(337) : wa
rning C4530: C++ exception handler used, but unwind semantics are not enabled. S
pecify /EHsc
scope2.cpp(15) : error C2065: 'k' : undeclared identifier
scope2.cpp(18) : error C2065: 'j' : undeclared identifier
```

Listing 6.4: 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.

6.1.3 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.

```
#include <iostream>

using namespace std;

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 = new 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
    cout << "Array 1" << endl;
    for (int i = 0; i < 20; ++i)
        cout << data[i] << endl;

    data = create_array_new();
    // Print out all elements
    cout << endl << "Array 2" << endl;
    for (int i = 0; i < 20; ++i)
        cout << data[i] << endl;
    // Free the memory
    delete[] data;
    // Set to nullptr
    data = nullptr;

    // Create array from pointer
    // This will cause a memory allocation error
    create_array(data);

    // Allocate memory
    data = new int[20];
    // Create array from pointer
    create_array(data);
    // Print out all elements
    cout << endl << "Array 3" << endl;
    for (int i = 0; i < 20; ++i)
        cout << data[i] << endl;
    // Free the memory
    delete[] data;
    // Set to nullptr
    data = nullptr;

    return 0;
}
```

Listing 6.5: 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 `nullptr` (which represents memory location 0). We will come back to `nullptr` later. 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 6.6: 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.

6.2 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 6.1 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.

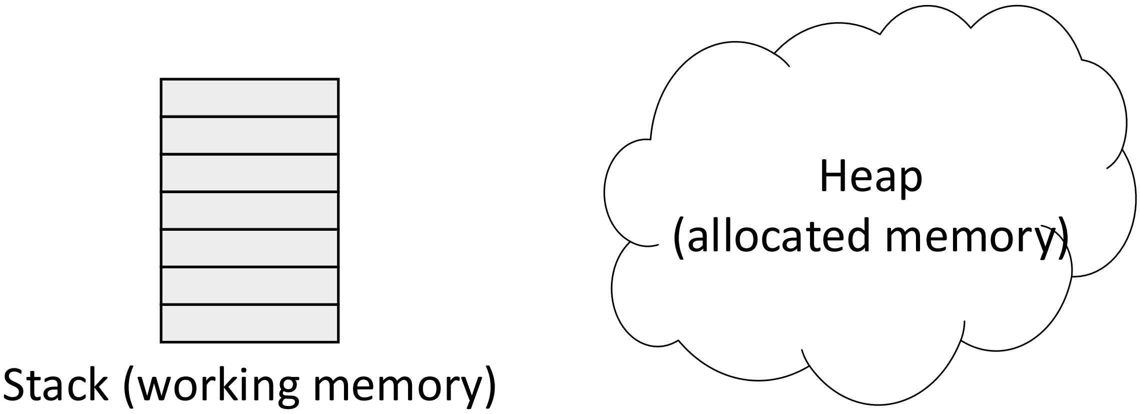


Figure 6.1: Stack and Heap

6.2.1 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. We will look into C++ allocation shortly. However, first we need to look at how we deallocate memory.

6.2.2 The free Function

In C and C++ we are usually responsible for allocating memory and ensuring that it is cleaned up afterwards. 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:

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

6.2.3 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`.

6.2.4 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`

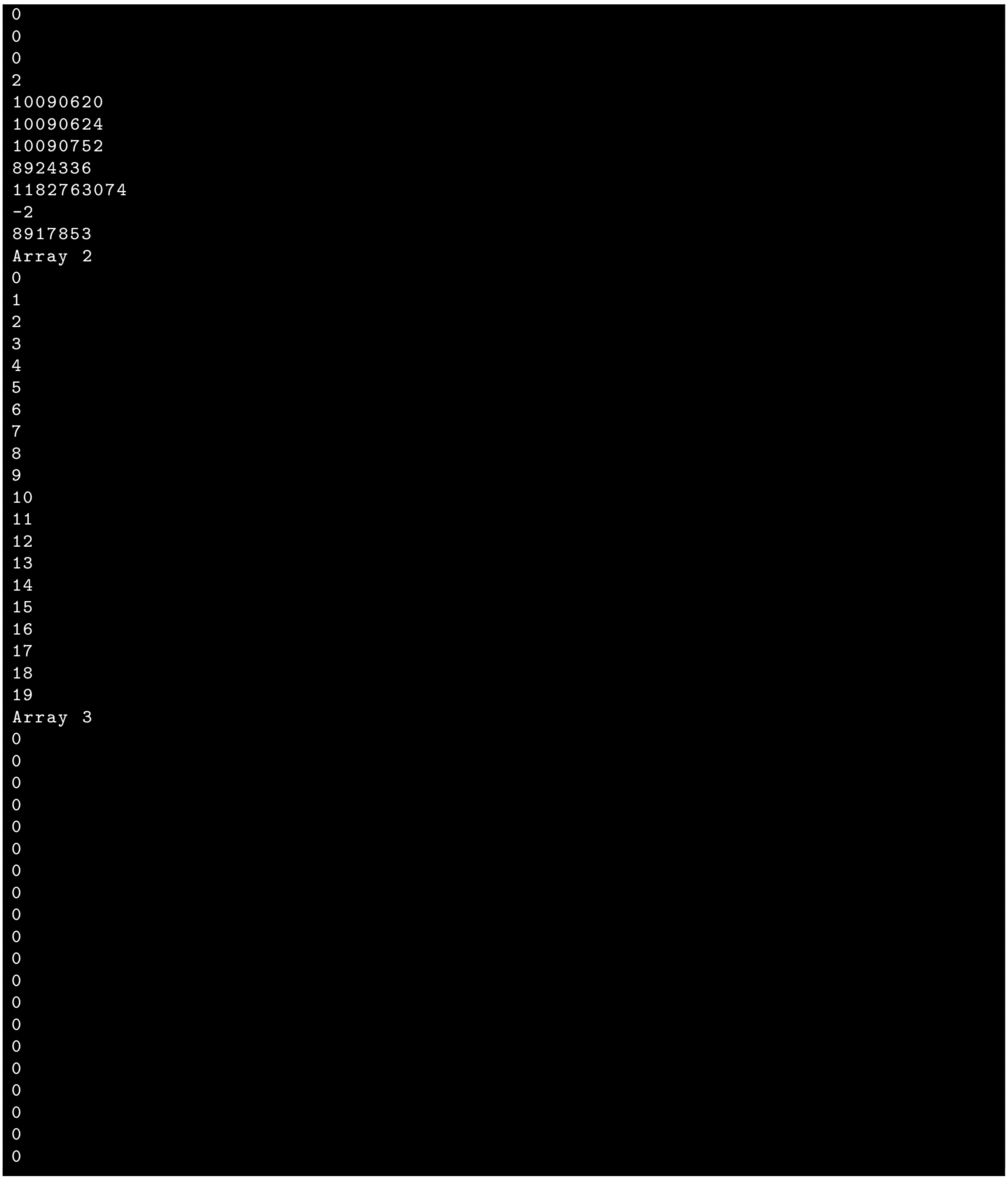
The example application is below.

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

int* foo()
{
    // 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* foo2()
{
    // 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
}

int* foo3()
{
    // Memory allocated on the heap using calloc
    // Memory set to 0.  Use elements and element size as seperate parameters
    int *data = (int*)calloc(20, sizeof(int));
    return data;
    // Memory on the heap still relevant
}

int main(int argc, char **argv)
{
    // Call foo - stack is corrupted
    int *data = foo();
    // Check values
    printf("Array 1\n");
    for (int i = 0; i < 20; ++i)
        printf("%d\n", data[i]);

    // Call foo2 - data on heap so not lost
    data = foo2();
    printf("Array 2\n");
    for (int i = 0; i < 20; ++i)
        printf("%d\n", data[i]);
    // Free memory
    free(data);

    // Call foo3 - data on heap so not lost
    data = foo3();
    printf("Array 3\n");
    for (int i = 0; i < 20; ++i)
        printf("%d\n", data[i]);
    // Free memory
    free(data);

    return 0;
}
```

Listing 6.7: 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 6.8: 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.

6.3 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. On Windows the default is 1 megabyte. This isn't that much really. This means that when you are working with any application that needs more that 1 megabyte 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 <cstdlib>
#include <cstdio>

// Size of data to allocate
const unsigned int SIZE = 1048576;

int main(int argc, char **argv)
{
    // Try and create data on the stack
    // This is 1MB - common stack size.
    // Other values take us over the stack size
    char data[SIZE];
    for (int i = 0; i < SIZE; ++i)
        printf("%d\n", data[i]);

    return 0;
}
```

Listing 6.9: Allocating Beyond the Limit of the Stack

When you compile and run this application you will get a runtime error (the application will crash). This is because we have attempted to allocate an array of 1 megabyte on the stack. The other values in scope will take us over 1 megabyte, hence the runtime error.

6.3.1 For the Brave - Setting the Stack Size with the Linker

As mentioned, 1 megabyte is the default stack size for Windows applications. You can however change the stack size for your application using the linker. Below is how we achieve this.

```
cl /c <filename.cpp>
link /STACK:<bytes> <filename.obj>
```

Listing 6.10: Setting the Stack Size with the Linker

See if you can get the above application to work by increasing the stack size to 1 megabyte. Also, try and find out if there is a limit to the stack size that you can define.

6.4 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 6.11: 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.

6.4.1 Exercise

Try and find the limit of the amount of memory you can allocate on both 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.

6.5 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.

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

void foo(int *value)
{
    // Allocate the value.  This changes the memory pointed to
    value = (int*)malloc(sizeof(int));
    *value = 5;
    // Print address and value
    printf("In first foo, address - %d, value - %d\n", value, *value);
    // When we return, we lose the memory address
    // We have a leak!!!
}

void foo(int **value)
{
    // Allocate the value.  This changes the memory pointed to
    *value = (int*)malloc(sizeof(int));
    **value = 5;
    // Print address and value - have to dereference twice
    printf("In second foo, address - %d, value - %d\n", *value, **value);
    // When we return, we retain the pointed to memory address
}

int main(int argc, char **argv)
{
    // Declare value.  Use calloc to set as 0
    int *value = (int*)calloc(1, sizeof(int));
    // Print address and value
    printf("Initially, address - %d, value - %d\n", value, *value);
    // Call foo
    foo(value);
    // Print address and value
    printf("After first foo, address - %d, value - %d\n", value, *value);
    // Call foo with pointer to pointer
    foo(&value);
    // Print address and value
    printf("After second foo, address - %d, value - %d\n", value, *value);
    // Free memory
    free(value);

    return 0;
}
```

Listing 6.12: 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 6.13: 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.

6.6 Using `new` in C++ to Allocate Memory

Allocating memory in C using `malloc` and `calloc` can be useful when working in C, but C++ provides a different mechanism - the `new` operator. `new` will also allocate memory on the heap but does so in a manner that is easier to read. In fact, you have used this method in Java already.

C++ memory location using `new` operates on the surface as C memory allocation using `malloc` and `calloc` does. Memory is still allocated in a block and we still need to free (or delete in C++ terms) the memory after we are finished with it. In the next section we will look at automatic memory management in C++ to make our life even easier.

6.6.1 Using `new` to Allocate Memory

`new` is not a function in C++ as `malloc` - it is an operator. Using `new` in C++ takes the following form for allocating a single value:

```
type *name = new type(paramaters);
```

Notice that we don't have to tell `new` the size of the value - it can work this out itself. We can also pass parameters (or the value) to the call to `new`. It is effectively the same call you would make in Java and C#. As an example, to allocate a new `int` with the value 5 we would use:

```
int *value = new int(5);
```

`new` as an operator

As mentioned `new` is not a function in C++ but an operator (like +, *, etc.). As such, a class can actually override the standard behaviour of `new` and undertake something different. This is not done too often, but can be used in particular circumstances when finer grained control of memory is required.

We will look at operator overloading (but not overloading `new` in the second half of the module.

6.6.2 Using `delete` to Free Memory

As `malloc` is paired with `free` in C, `new` is paired with `delete`. `delete` is also an operator and as such the call is very simple.

```
delete name;
```

We just use the variable name that we allocated to and work accordingly. For example, to delete the variable we allocated using `new` above we use the following:

```
delete value;
```

`delete` as an operator

As with `new`, `delete` is also an operator that can be overridden by the programmer for different classes. If you ever need to override `new` then you should also override `delete`. Again, we will come to operator overloading (but not of `delete`) later in the module.

6.6.3 Using new to Allocate Arrays

So we have seen how to use new to allocate a single value. What about arrays of values? Well in C++ we can do this in a similar manner to the one you have experienced in Java:

```
type *name = new type[size];
```

Again, we don't need to pass the size of the individual type - C++ can work this out for us. It just needs to know how many items you wish to allocate. As an example, to allocate 100 float values we would do the following:

```
float *values = new float[100];
```

Arrays in C and C++ Revisited

Notice that we are still just working with a pointer (single *) when using new when working with an array or a single value. Remember from our discussion before that an array is just a memory location where we state that a certain type of data is. In C++ we don't even know how many values there are in that memory location - we have to keep track of this ourselves.

When working with arrays in C and C++ it is worth remembering that we can access individual values as an offset from this base address. For example, if we have 10 values of type int (assuming int is 4 bytes) then the offset of each value of the array from the starting address is:

index	0	1	2	3	4	5	6	7	8	9
offset	0	4	8	12	16	20	24	28	32	36

When we used malloc we did not really specify that we might be only allocating a single value in memory - but it is possible. In C++ the distinction is more obvious because of the different syntax used.

6.6.4 Freeing Arrays with delete[]

In C++ we have to specify that we are deleting an array when calling delete. The call is as follows:

```
delete[] name;
```

Otherwise the idea is the same. For example, to delete our 100 float values used above we undertake the following:

```
delete[] values;
```

6.6.5 Test Application

Let us now build a test application for using new and delete. This is shown below. It follows the same pattern we used for malloc.

```
 1 #include <iostream>
 2 
 3 using namespace std;
 4 
 5 int* foo()
 6 {
 7     // Create memory on the heap
 8     int *data = new int(5);
 9     // Return data
10     return data;
11 }
12 
13 int* foo2(unsigned int size)
14 {
15     // Create an array on the heap
16     int *data = new int[size];
17     // Return data
18     return data;
19 }
20 
21 int main(int argc, char **argv)
22 {
23     // Call foo
24     int *data = foo();
25     // Output address and value
26     cout << "Address = " << data << ", value = " << *data << endl;
27     // Delete the memory
28     delete data;
29 
30     // Call foo2
31     int *data2 = foo2(20);
32     // Output addresses and value
33     for (int i = 0; i < 20; ++i)
34         cout << "Array address = " << &data[i] << ", value = " <<
               data[i] << endl;
35     // Delete the memory
36     delete[] data2;
37 
38     return 0;
39 }
```

Listing 6.14: Using `new` and `delete`

Notice on line 34 we are also printing the *address of* the array elements. This is to illustrate that the array elements are continuous in memory. An output for this application (depending on the memory addresses used) is shown below:

```
Address = 0084A928, value = 5
Array address = 0084A928, value = 8694344
Array address = 0084A92C, value = 8650944
Array address = 0084A930, value = 232176122
Array address = 0084A934, value = 134221306
Array address = 0084A938, value = 14541700
Array address = 0084A93C, value = 1
Array address = 0084A940, value = 81181171
Array address = 0084A944, value = 134221306
Array address = 0084A948, value = 8695168
Array address = 0084A94C, value = 8694408
Array address = 0084A950, value = 0
Array address = 0084A954, value = 0
Array address = 0084A958, value = 8694044
Array address = 0084A95C, value = 0
Array address = 0084A960, value = 8694040
Array address = 0084A964, value = 0
Array address = 0084A968, value = 0
Array address = 0084A96C, value = 14579872
Array address = 0084A970, value = 0
Array address = 0084A974, value = 0
```

Listing 6.15: Output from C++ Memory Allocation Application

Notice the memory addresses for arrays. Our array has memory address 0084A928 (in hexadecimal). This is also the address of the first element (element 0). Every other element has this base address plus four times its own index. So element index 7 has the base memory address plus 28 (so memory location 0084A944 in hex). Understanding this relationship between memory addresses and items in an array is also an important concept when dealing with low level memory concerns.

6.7 Best Practice When Working with Allocated Memory

We have actually been very bad in our current use of memory. When deleting or freeing memory we should always set our memory to `null` afterwards. In raw memory terms this means setting the pointer to 0 (equivalent to not allocated). In C we should do the following:

```c
void *mem = malloc(1024);
// Do something with the memory
free(mem);
// Set pointer to null (0)
mem = 0;
```

In C++ we would do something like this:

```cpp
int *x = new int(5);
// Do something with x
delete x;
x = 0;
```

In modern C++ (C++11 onwards) we can use the `nullptr` value (which also equals 0):

```cpp
int *x = new int(5);
// Do something with x
delete x;
x = nullptr;
```

From this point onwards you should always free your memory and then set it to ***null****.* In particular, when working with C++ you should use `nullptr` as this is the standard. Remember this as it can solve a number of issues. We will also return to `nullptr` in the next few sections.

6.8 shared_ptr and Automatic Memory Management

Working with memory has been a major issue for new (and even experienced) C and C++ programmers. Thankfully the C++11 standard introduced new data types that overcome many of the issues encountered when working with memory - *smart pointers*. There are two types of smart pointers - `shared_ptr` (pronounced *shared puter*) and `unique_ptr` (*unique puter*). We will look at `shared_ptr` first.

Garbage Collection

In languages and runtimes such as Java, C#, Python and JavaScript our memory management is handled for us. This is done using a technique called `garbage collection`. Garbage collection means that the runtime takes care of determining when memory is being used and when it can be freed. This makes the life of a programmer *much easier*. However, it has an impact on performance - the tracking of valid and invalid memory and the freeing of it takes processor time.

From the advent of Java many people have considered garbage collection the best method for most applications. However, the introduction of smart pointers have driven a vocal argument about garbage collection being a problem. At the end of the day, unless you are writing high performance, well optimised code, there is really no argument against garbage collection - you should always use a language that is best fit for the purpose (no language is good in all conditions). However, understanding when your values are valid and not in memory is an important skill.

Working with `shared_ptr`

`shared_ptr` tries to work as much like a standard pointer as possible. This means that much of our previous examination of pointers in C++ carries on to `shared_ptr`. Many of the differences come from declaration and creating the `shared_ptr`.

To declare a `shared_ptr` we use the following:

```
shared_ptr<type> name;
```

The `type` parameter tells us what is pointed to. So for example to create a `shared_ptr` of type `int` we do the following:

```
shared_ptr<int> value;
```

To actually create a `shared_ptr` (remember that declaring a pointer doesn't mean allocating memory) we need to pass a pointer as a parameter:

```
shared_ptr<type> name = shared_ptr<type>(new type(params...));
```

So to create a `shared_ptr` of type `int` we would use the following:

```
shared_ptr<int> value = shared_ptr<int>(new int(5));
```

Another method of creating a `shared_ptr` is using the `make_shared` function. This function works out the correct method to create the object by the parameters passed to it. So for an `int` we can use the following:

```
shared_ptr<int> value = make_shared(5);
```

The `make_shared` function is the better method for creating `shared_ptr` values in general as it involves no copying of data.

The shared_ptr value behaves much like a standard pointer otherwise. When the dereference operator (*) is used we can access the value. There are other operations on shared_ptr but the only one we will look at just now is getting the usage count. This is done using the use_count method on the shared_ptr. This tells us how many references there are to the pointer. When the shared_ptr is passed as a parameter (enters a new scope) the use count is incremented. Exiting a scope or setting one of the shared_ptr to nullptr decrements the count. When the count equals 0 the memory is deleted.

Let us build an example application using shared_ptr. This is shown below:

```cpp
#include <iostream>
#include <memory>

using namespace std;

void foo(shared_ptr<int> value)
{
    // Print out number of accessors to the shared object - should be 2
    cout << "In foo, count on shared_ptr = " << value.use_count() << endl;
    // shared_ptr behaves like a pointer - we can assign to it by dereferencing
    *value = 5;
}

int main(int argc, char **argv)
{
    // We can create a shared_ptr by wrapping it around an allocated object
    shared_ptr<int> value = shared_ptr<int>(new int(10));
    cout << "Initial address = " << value << ", value = " << *value << endl;
    // We can also create a shared_ptr by using the make_shared function
    value = make_shared<int>(70);
    cout << "After make shared, address = " << value << ", value = " << *value << endl;
    // Print out number of accessors to the shared object
    cout << "Before foo, count on shared_ptr = " << value.use_count() << endl;
    // Let's call foo
    foo(value);
    cout << "After foo, address = " << value << ", value = " << *value << endl;
    // Print out number of accessors to the shared object
    cout << "After foo, count on shared_ptr = " << value.use_count() << endl;
    // Let's set value to nullptr - this will clear the memory
    value = nullptr;
    // Print out number of accessors to the shared object.  Should be 0
    cout << "After nullptr, count on shared_ptr = " << value.use_count() << endl;

    return 0;
}
```

Listing 6.16: Using shared_ptr

What is nullptr?

We already mentioned that nullptr can be used to set a pointer to null. However, it does more than this. nullptr was introduced with smart pointers. It allows us to set a smart pointer to null. nullptr can be used as a shared_ptr value which 0 does not. However, nullptr can be used as 0. Therefore, nullptr should be used wherever possible.

Referencing Counting of Pointers

The idea of referencing counting of pointers has been around for a long time. Almost every company that used C++ extensively created their own version of a reference counting data type. C++11 introduced this as a language feature of C++ rather than a user defined type. This has changed how C++ should be approached as a programming language. In general, smart pointers are the method you should use from now on.

Using make_shared

make_shared is a useful C++ function which uses some C++ magic to work - magic we won't look at in this module. If you are interested then looking at *variadic templates* and *metatemplate programming* is a worthwhile area of research (after the module though).

make_shared works by calling the necessary *constructor* (more on this when we discuss object-orientation). It works this out by the parameters passed as the compiler working out the correct method to call. If no constructor exists with those parameters we get a compiler error.

The following is an example output from the application:

```
Initial address = 00D3A948, value = 10
After make shared, address = 00D3AA7C, value = 70
Before foo, count on shared_ptr = 1
In foo, count on shared_ptr = 2
After foo, address = 00D3AA7C, value = 5
After foo, count on shared_ptr = 1
After nullptr, count on shared_ptr = 0
```

Listing 6.17: Output from shared_ptr Application

The use count value illustrates how reference counting works. This is the important aspect of shared_ptr. We can pass the value around happily and when we no longer need it the memory is cleared up. Our other smart pointer - unique_ptr - is not for sharing. It is only valid in one scope.

6.9 Examining shared_ptr

As with a normal pointer we cannot just pass in shared_ptr values as parameters and change the pointer. When passing a shared_ptr value into a function a copy of the shared_ptr is made (but not a copy of the pointed to data). This means that changing the address (for example using make_shared) changes the pointer. This is the same problem we had to overcome using pointers-to-pointers.

To overcome this we can use a reference to our shared_ptr. The following application illustrates this:

```
 1 #include <iostream>
 2 #include <memory>
 3 
 4 using namespace std;
 5 
 6 void foo(shared_ptr<int> value)
 7 {
 8     // Output number of accessors to value - should be 2
 9     cout << "In foo, accessors to value = " << value.use_count() <<
          endl;
10     // Call make_shared again - we lose this copy
11     value = make_shared<int>(25);
12     // Output number of accessors to value - should be 1
13     cout << "In foo, after make_shared, accessors to value = " <<
         value.use_count() << endl;
14     // No memory leaks! shared_ptr will clean itself up
15 }
16 
17 // Note we now have a reference to the shared_ptr
18 void foo2(shared_ptr<int> &value)
19 {
20     // Output number of accessors to value - should be 1
21     cout << "In foo2, accessors to value = " << value.use_count()
         << endl;
22     // Call make_shared again - we retain this copy
23     value = make_shared<int>(100);
24     // Output number of accessors to value - should still be 2
25     cout << "In foo2, after make_shared, accessors to value = " <<
         value.use_count() << endl;
26 }
27 
28 int main(int argc, char **argv)
29 {
30     // Create a new shared_ptr
31     shared_ptr<int> value = make_shared<int>(5);
32     // Print data
33     cout << "Address = " << value << ", value = " << *value << ",
         usage = " << value.use_count() << endl;
34     // Call foo
35     foo(value);
36     // Print data
37     cout << "Address = " << value << ", value = " << *value << ",
         usage = " << value.use_count() << endl;
38     // Call foo2
39     foo2(value);
40     // Print data
41     cout << "Address = " << value << ", value = " << *value << ",
         usage = " << value.use_count() << endl;
42 
43     return 0;
44 }
```

Listing 6.18: Examining the shared_ptr Value

Below is an example output. Notice that the printing of the value on line 13 illustrates that the use count is changed to 1 after the call to make_shared. This is the problem that foo2 overcomes by passing the shared_ptr as a reference.

```
Address = 00A9A964, value = 5, usage = 1
In foo, accessors to value = 2
In foo, after make_shared, accessors to value = 1
Address = 00A9A964, value = 5, usage = 1
In foo2, accessors to value = 1
In foo2, after make_shared, accessors to value = 1
Address = 00A9AACC, value = 100, usage = 1
```

Listing 6.19: Output from Examing the shared_ptr Value

6.10 Using shared_ptr for Arrays

shared_ptr works well for standard values allocated in the heap. For arrays it is a little different. This is because of the different allocation and deletion methods that arrays use. As such, when working with arrays and shared_ptr we have to declare them a little differently.

Firstly, the make_shared function cannot be used for arrays - there is no object constructor for arrays. Therefore we always have to use the standard shared_ptr constructor:

```
shared_ptr<type> name = shared_ptr<type>(new type[size]);
```

So to create an array of size 10 we would use the following:

```
shared_ptr<int> value = shared_ptr<int>(new int[10]);
```

This is only one part of the problem. shared_ptr does not know that our value is an array and therefore won't delete the memory properly. Thus we need to tell shared_ptr how to delete the memory. We do this by passing a deletion method. C++ comes with some default ones that we can use for arrays:

```
shared_ptr<type> name = shared_ptr<type>(new type[size], default_delete<type[]>());
```

So for our int array we can use the following:

```
shared_ptr<int> name = shared_ptr<int>(new int[10], default_delete<int[]>());
```

Below is an example application to show how we do this:

```cpp
#include <iostream>
#include <memory>

using namespace std;

const int SIZE = 65536;

void build_array(shared_ptr<int> data)
{
    for (int i = 0; i < SIZE; ++i)
        // Have to dereference the pointer, then access the values
        data.get()[i] = i;
}

void print_array(shared_ptr<int> data)
```

```
16 {
17     for (int i = 0; i < SIZE; ++i)
18         // Have to dereference the pointer, then access the values
19         cout << data.get()[i] << endl;
20 }
21
22 int main(int argc, char **argv)
23 {
24     // We have to tell shared_ptr how to delete the data
25     // When creating use the default_delete function
26     shared_ptr<int> data = shared_ptr<int>(new int[SIZE],
           default_delete<int[]>());
27     // Build the array
28     build_array(data);
29     // Print the array
30     print_array(data);
31     // Set data to nullptr - will cause data to be deleted
32     data = nullptr;
33
34     return 0;
35 }
```

Listing 6.20: Using shared_ptr for Arrays

On line 12 we use the get method on the shared_ptr. This allows us to get the raw pointer controlled by the shared_ptr. We need to do this to access the individual elements of the array. We cannot do this via the shared_ptr - it is not an array. Therefore we have to get the raw pointer and work with it in the way we want.

6.11 unique_ptr

shared_ptr is a good replacement for almost every situation where we would use a pointer. However, it comes with an overhead for the management of the reference counting. It is also the case that sometimes we don't need to share the pointer - there is only ever one owner. This is true for when we have to allocate large temporary blocks of memory inside functions. The unique_ptr is a solution to this.

Why we need unique_ptr

unique_ptr works by monitoring which scope currently owns the pointer. The pointer can only be owned by one function. When you pass the unique_ptr to another function the ownership is passed to this function and the caller loses ownership.

This idea can be quite tricky for new programmers to understand. C++11 introduced the idea of *move semantics*. Move semantics are an area we won't cover in this module any further, but you should look into these further if you are to become a C++ programmer. Move semantics mean that we *move* data from one function to another. No copying takes place, but also no reference passing occurs. This is the middle ground between pass-by-value and pass-by-reference.

We need unique_ptr for situations where we don't need reference counting. In general you probably should stick with shared_ptr unless you are sure you don't need pointer sharing. This takes experience and thought to determine.

Difference between unique_ptr and shared_ptr

The difference between shared_ptr and unique_ptr is the reference counting. unique_ptr has none - it uses move semantics to ensure that only one owner of the pointer exists. When that owner goes out of scope the memory is cleared up.

shared_ptr adds to the use count every time a copy (pass-by-value) of the pointer is made. The use count is decremented every time one of these copies goes out of scope. When the use count equals 0 the memory is cleared up.

Using shared_ptr and unique_ptr is now the common practice in C++. Working with raw memory should only occur when working at a low level such as using hardware. This makes working with C++ easier for new programmers - once they understand using smart pointers.

An example application of using unique_ptr is below. Note the problems encountered when passing the data to a function without using a reference. Also, there is currently no make_unique function as there is make_shared supported by the Microsoft compiler - although the C++14 standard does introduce this.

```
#include <iostream>
#include <memory>

using namespace std;

void foo(unique_ptr<int> value)
{
    // Let's set the value to 500.  Can treat just as a pointer
    *value = 500;
    // We don't return unique_ptr.  We have lost *ALL* the data
}

unique_ptr<int> foo2(unique_ptr<int> value)
{
    // This time we will move the answer back
    *value = 500;
    return value;
}

void foo3(unique_ptr<int> &value)
{
    // Using a reference solves the problem
    *value = 1000;
}

int main(int argc, char **argv)
{
    // Create a unique_ptr value
    unique_ptr<int> value(new int(50));
    // Print data
    cout << "Address = " << value.get() << ", value = " << *value
        << endl;
    // Call foo.  We have to "move" the pointer into foo
    foo(move(value));
    // Print data.  Cannot print value.  No longer valid
    cout << "After foo, address = " << value.get() << endl;

    // Recreate the value
38      value = unique_ptr<int>(new int(50));
39      // Print data
40      cout << "Address = " << value.get() << ", value = " << *value
            << endl;
41      // Call foo2.  We have to "move" the pointer into foo
42      value = foo2(move(value));
43      // Print data
44      cout << "After foo2, address = " << value.get() << ", value = "
             << *value << endl;
45
46      // Call foo3.  Use a reference to avoid a move
47      foo3(value);
48      // Print data
49      cout << "After foo3, address = " << value.get() << ", value = "
             << *value << endl;
50
51      return 0;
52  }
```

Listing 6.21: Using unique_ptr

On line 20 we overcome the move of a unique_ptr by using pass-by-reference. Passing smart pointers by reference overcomes many issues that we had to use pointers-to-pointers. An example output from this application is below:

```
Address = 0107A948, value = 50
After foo, address = 00000000
Address = 0107A948, value = 50
After foo2, address = 0107A948, value = 500
After foo3, address = 0107A948, value = 1000
```

Listing 6.22: Output from unique_ptr Application

6.12 unique_ptr for Arrays

Unlike shared_ptr, unique_ptr can happily work with arrays. As unique_ptr is meant for situations where we need blocks of temporary memory this makes sense. Below is an example application showing how we use arrays with unique_ptr.

```
#include <iostream>
#include <memory>

using namespace std;

const int SIZE = 65536;

void build_array(unique_ptr<int[]> &data)
{
    for (int i = 0; i < SIZE; ++i)
        // Have to dereference the pointer, then access the values
        data.get()[i] = i;
}

void print_array(unique_ptr<int[]> &data)
{
    for (int i = 0; i < SIZE; ++i)
        // Have to dereference the pointer, then access the values
        cout << data.get()[i] << endl;
}

```

```
int main(int argc, char **argv)
{
    // unique_ptr can have arrays
    unique_ptr<int[]> data(new int[SIZE]);
    // Build the array - using pass by reference on the unique_ptr
    build_array(data);
    // Print the array
    print_array(data);
    // Set data to nullptr - will delete the array
    data = nullptr;

    return 0;
}
```

Listing 6.23: Using unique_ptr for Arrays

Notice that we still have to declare the array using new and pass this to the unique_ptr. Otherwise we don't have to do anything else special to work with arrays and unique_ptr.

6.13 The auto Keyword

We are now getting to a point where we are getting quite complicated type names. For shared_ptr, unique_ptr and vector we can end up writing quite long type names. For example, we could easily have a type of the following:

```
vector<shared_ptr<vector<int>> data;
```

This is a vector containing a collection of shared_ptr to vector of int - or a multidimensional array like structure using vector. This is actually not an uncommon thing to do in large applications.

Writing out this type name can be tiresome. Thankfully C++11 introduced the new auto keyword. This allows the compiler to determine the type for us. An example application is below.

```
#include <iostream>
#include <memory>

using namespace std;

int main(int argc, char **argv)
{
    // Use the auto keyword - compiler determines the type
    auto data = make_shared<int>(5);
    // ... data is a shared_ptr<int>.  Compiler fills in the type
    return 0;
}
```

Listing 6.24: Using auto to Simplify Code

Using the auto keyword

The auto keyword for a data type only works when the compiler can work out the type. For example we cannot simply write the following:

```
auto x;
```

The compiler has no way of determining the type - `x` could be anything. Changing this to the following:

```
auto x = 5;
```

Means that `x` is determined to be of type `int`.

The `auto` keyword comes in very handy as you get further into C++ programming. In C++14 it can also be used for parameters which is pretty powerful. At the moment though you should only use it for variable declarations.

The `auto` keyword should not be confused with dynamic typing seen in languages such as JavaScript and Python. The compiler is determining the type of your variable and this type cannot be changed. For example, it is not OK to write the following:

```cpp
// Type is shared_ptr<int>
auto x = make_shared<int>(5);
// Now just try and set x to 5 (so just an int)
x = 5;
// This will cause a compiler error
```

`auto` does not enable dynamic typing in C++ - due to it's low level nature this is not possible. Don't think of `auto` as any type. It is just a method to let the compiler fill in the type for us.

6.14 For the Brave - Casting in C++ Using `static_cast`

For the last part of this unit we will look at some of the C++ casting methods. We already looked at casting from a C point of view, and this still exists in C++. However, this type of casting is frowned upon in C++ and other techniques should be used instead.

Remember our discussion on C casting. From that point of view we were simply stating that an area of memory should be treated as a different type. This has no checking involved which can lead to a number of problems. C++ uses the `static_cast` function to allow us cast between types in C++ safely. It takes the following form:

```
type name = static_cast<type>(value);
```

So to cast a `float` to an `int` we would do the following:

```cpp
float x = 12.5f;
int y = static_cast<int>(x);
```

Let us look at an example application where we try and cast a `string` to a `float`. This code is shown below:

```cpp
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char **argv)
{
  // Declare an int
```

```
 9   int x = 10;
10   // Cast to a float
11   // Let compiler work out the type
12   auto y = static_cast<float>(x);
13   // Declare a string
14   string str = "Hello world!";
15   // Cast str to a float
16   // Will cause a compiler error
17   auto z = static_cast<float>(str);
18
19   return 0;
20 }
```

Listing 6.25: Using static_cast

Compiling this application will provide a compiler error as shown below. Notice that the line (17) is given that is the problem and the issue (no conversion operator available).

```
cl static_cast.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.30501 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

static_cast.cpp
C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\INCLUDE\xlocale(337) : wa
rning C4530: C++ exception handler used, but unwind semantics are not enabled. S
pecify /EHsc
static_cast.cpp(17) : error C2440: 'static_cast' : cannot convert from 'std::str
ing' to 'float'
        No user-defined-conversion operator available that can perform this conv
ersion, or the operator cannot be called
```

Listing 6.26: Compiler Output from Casting Error

6.15 For the Brave - Casting with const_cast

Another casting function C++ provides is const_cast. const_cast allows us to convert a const value to a non-const value. This can overcome a situation where we need to modify a const value.

Let us look at an example application where we cast away the const of the internal char array in a string. This allows us to modify the individual characters in the string. This example application is below.

```
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char **argv)
{
  // Create a string
  string str = "Hello World!";
  cout << "String = " << str << endl;
  // Get the raw C string
  const char *const_raw = str.c_str();
  // Can't change any values in raw - const
  // This won't compile
  // const_raw[4] = 'p';
  // Now let's live dangerously - use const cast
  char *raw = const_cast<char*>(const_raw);
  // Change the value of raw now
  raw[4] = 'p';
  // Print out the string
  cout << "String = " << str << endl;

  return 0;
}
```

Listing 6.27: Using `const_cast`

In this application we are changing the 5th character of “Hello World!” (remember that the 1st character is index 0) to “p” giving us “Hellp World!”. The output is shown below.

```
String = Hello World!
String = Hellp World!
```

Listing 6.28: Output from Manipulating `const` Values with `const_cast`

WARNING - avoid `const_cast`

So we can remove the `const` of a value - surely that can be useful. Well yes, but if you have written an application that requires you to do this you should probably rethink what you have done. Values are normally declared `const` for a reason and removing this normally circumvents this requirement. Rethink your application if you find that you require `const_cast`.

6.16 For the Brave - Using typeid

Although we are generally just dealing with raw memory blocks in C++ and just stating what we want to treat the memory as, it is possible to get type information in C++. It isn't that common as it does have an overhead, but we can use the `typeid` function to get a value's type. This type is itself a type with some values and operations associated with it.

Below is a test application using `typeid`. This is all we will do with types in C++ apart from working with object-orientation concepts. However, you should consider typing a fundamental part of programming.

```
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char **argv)
{
  // Create a string
  string str1 = "Hello";
  string str2 = "World";

  // Check if the types are equal
  if (typeid(str1) == typeid(str2))
    cout << "Types are equal" << endl;
  else
    cout << "Types are not equal" << endl;

  // Create a float
  float x = 52.0f;

  // Print out details
  cout << "x is a " << typeid(x).name() << " with value " << x << endl;

  return 0;
}
```

Listing 6.29: Using `typeid`

We are testing if two values of type `string` have the same type (they do), and then getting the name of the `float` type. The output from this application is shown below:

```
Types are equal
x is a float with value 52
```

Listing 6.30: Output from `typeid` Application

6.17 Case Study - Building Trees

This exercise is quite long and in a number of stages. *It is strongly advised that you undertake this exercise as it will really help in your understanding of these concepts.* The basic parts covered are useful for understanding, and the extra exercises will really push the capabilities of the stronger students in the class.

6.17.1 Trees

Not the type you find in a park but a data structure used to efficiently store and access data. You will come across trees in a number of places in computing, such as data structures, network architectures, etc. We will introduce a basic type of tree - the binary tree - and how we can use that to store ordered data.

An example of a binary tree is given in Figure 6.2. Let us define a few terms we will use.

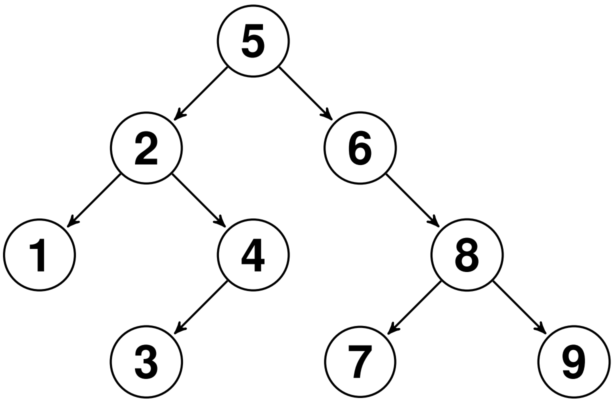


Figure 6.2: Binary Tree Example

node - a node is an item in the tree (one of the circles with a number in). A tree is just a collection of nodes connected together in a particular manner.

root node - the top node of the tree being examined.

branch - a branch connects a node to other nodes. In a binary tree each node has two branches - *the left branch* and *the right branch*.

A binary tree stores data in manner such that the value store on the left branch of a node is always less than the value store in the node, and the branch on the right contains values that are larger than the value in the node. This allows us to efficiently store data in order and retrieve it.

This also brings up an idea that you also need to grasp - *a branch of a tree is itself a tree*. Think about it. If we take a branch, then we are effectively considering the root node as the top node of that branch. This allows us to think of the tree as a collection of sub-trees, which comes in handy when we start dealing with our algorithms.

Exercise

Check the tree in Figure 6.2 now. Look at each node and then look at the node on its left branch and then its right branch. Is the tree correct? Discuss with someone else in the class and then one of the teaching team to determine its correctness.

6.17.2 Building a Binary Tree

Next let us examine how we created the tree in Figure 6.2. We will look at the left hand side first. At the start, the tree is obviously empty. The first value added to the tree is 5. This gives us the tree shown in Figure 6.3.

Figure 6.3: Binary Tree Start

Now let us insert the values 2, 4, 1, and 3 into the tree. The sequence of steps involved in doing this is shown in Figure 6.4.

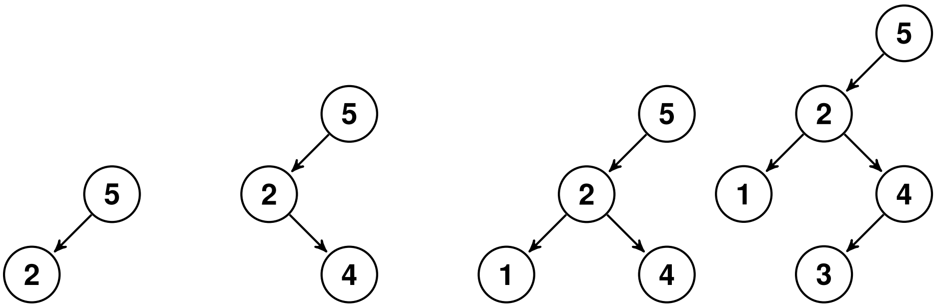


Figure 6.4: Building the Left Branch

Notice the stages of the tree building. When we add a new value we check if it against each node to see if it should go left or right until we find an empty place to add it. We will come back to this algorithm shortly.

Exercise

Go through the stages of adding the values 6, 8, 9 and 7. Do you get the same tree as shown in Figure 6.2?

6.17.3 Getting Started with Binary Trees

We know enough now to build a binary tree data structure and also some algorithms to enable testing of the binary tree. First of all, let us consider what a binary tree is in a programming context. Well a binary tree is just a collection of nodes which have branches to connect them. These branches connect to other nodes. Therefore we can consider a node as a data value (the value stored in the node) and pointers to left and right branches. Figure 6.5 illustrates the general idea.

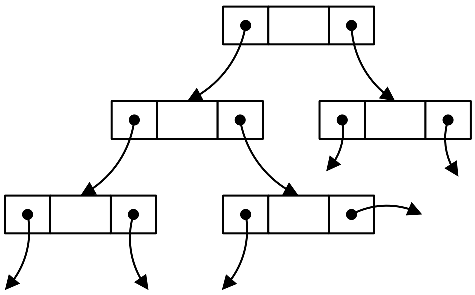


Figure 6.5: Data Structure of a Binary Tree

Therefore we can define a binary tree as just a node, with pointers to two other nodes. In code we will define it as follows:

```
struct node
{
    // Data stored in this node of the tree
    int data;
    // The left branch of the tree
    node *left;
    // The right branch of the tree
    node *right;
};
```

Listing 6.31: Binary Tree Node

Our application is going to perform three different techniques at present:

1. Insert a value into the tree

2. Print the tree *in order*

3. Free the resources in the tree

This requires us to utilise *recursion* in our code. We need to treat each branch as a tree, and we just drill down these trees until we get to the bottom.

What is Recursion?

Recursion is one of those ideas in programming that can throw the beginner until they change their way of thinking to accommodate. Recursion is where we call a function from within itself to allow certain types of computation to occur. Recursion is very common in algorithmic design, and in particular in functional programming (for those of you doing Mathematics for Software Engineering).

The basic concept is as follows. Let us look at the Fibonacci number sequence. It is defined as follows:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Notice that starting from 2, each number is the sum of the two previous ($2 = 1 + 1, 3 = 2 + 1, 5 = 3 + 2$, etc.). We can generalise this to the following equation:

$$F_n = F_{n-1} + F_{n-2}$$

With starting values $F_1 = 1$ and $F_2 = 1$.

If we were to write this out as code we would need to use the Fibonacci sequence function to define the Fibonacci sequence. The pseudocode for this would look as follows:

```
function FIBONACCI($n$)
    if $n = 1$ then
        return 1
    else if $n = 2$ then
        return 1
    else
        return FIBONACCI($n - 1$) + FIBONACCI($n - 2$)
```

As an exercise, use the above functional definition to determine the 10th Fibonacci number.

Our starting code for the exercises is below. You should be able to compile and run this code as is at the moment, although it won't do anything.

```cpp
#include <iostream>

using namespace std;

// The node of a tree
struct node
{
    // Data stored in this node of the tree
    int data;
    // The left branch of the tree
    node *left;
    // The right branch of the tree
    node *right;
};

// Inserts a value into the tree - notice **
void insert(node **tree, int value)
{
    // TODO
}

// Deletes the tree - freeing memory
void delete_tree(node *tree)
{
    // TODO
}

// Prints the tree in order
void inorder(node *tree)
{
```

```
31     // TODO
32 }
33 
34 int main(int argc, char **argv)
35 {
36     // Declare the tree - nullptr to a node
37     node *tree = nullptr;
38 
39     // Loop until -1 entered
40     while (true)
41     {
42         int num;
43         cout << "Enter number (-1 to exit): ";
44         cin >> num;
45 
46         if (num == -1)
47             break;
48         insert(&tree, num);
49     }
50 
51     // Print the tree
52     inorder(tree);
53     cout << endl;
54 
55     // Delete the tree
56     delete_tree(tree);
57 
58     return 0;
59 }
```

Listing 6.32: Binary Tree Application Starting Code

Notice on line 37 we just declare our tree as a pointer to a `node` which we initially set as `nullptr`. This just means that our initial tree is empty. The rest of the code is to prompt the user for a number to add to the tree and insert it. If the number entered is -1 we exit the loop, print the tree, then delete it.

Also note that our `insert` operation takes a pointer-to-pointer. This is because we will be allocating new `node` values here.

6.17.4 Inserting a Value into a Empty Tree

Let us start with the simplest idea - inserting a value into an empty tree. For this first part you are provided with the code:

```
// Inserts a value into the tree - notice **
void insert(node **tree, int value)
{
    // Check if nullptr.  If so set new node
    if (*tree == nullptr)
    {
        // Create new node
        *tree = new node;
        // Set new value
        (*tree)->data = value;
        // Set branches to nullptr
        (*tree)->left = nullptr;
        (*tree)->right = nullptr;
    }
}
```

Listing 6.33: Inserting into an Empty Tree

Lines 10, 12 and 13 need a little explaining. As we have a pointer-to-pointer situation, we need to dereference the pointer to access the value. We could do this twice (`**tree.data = value`), but we can also use the *pointed to* operator `->`. This operator can replace a single dereference and dot. Therefore the code `(*tree)->data` is equivalent to `(**tree).value`. Good practice in C and C++ normally means we use the pointed to operator when we can.

Exercise

You should be able to run the application now and add a single item to the tree. Try this now. At the moment you cannot really see if it worked, but you are checking that the application compiles and runs correctly.

6.17.5 Challenge 1 - Printing In Order

The `inorder` function allows the printing of the tree data in numerical order. This will allow use to test our application. The pseudocode for this operation is given below. Note the use of recursion.

```
procedure INORDER(tree)
    if tree ≠ nullptr then
        INORDER(left branch)
        PRINT(value)
        INORDER(right branch)
```

Your challenge here is to implement this algorithm and then use it to test the insertion we build above.

6.17.6 Challenge 2 - Inserting into a Non-Empty Tree

Now let us expand the `insert` operation to deal with non-empty trees. The complete algorithm for `insert` is given below. Note the use of recursion.

```
procedure INSERT(tree, value)
    if tree = nullptr then
        Create new tree
        data ← value
        Set left and right branches to nullptr
    else
        if value < data then                    ▷ is less than this node - go left
            INSERT(left branch, value)
        else if value > data then               ▷ is greater than this node - go right
            INSERT(right branch, value)
        else                                    ▷ is equal to this node - ignore
            return
```

When this is working and you test your application you will be able to enter as many numbers as you like. When `inorder` is called the numbers entered are printed in numerical order.

6.17.7 Challenge 3 - Freeing Resources

OK, we have been avoiding freeing up our memory and this is bad. Let us rectify this by completing the `delete_tree` operation. The pseudocode for this is below. Again, notice the recursion.

```
procedure DELETE_TREE(tree)
    if tree = nullptr then
        return
    DELETE_TREE(left branch)
    Delete left branch
    DELETE_TREE(right branch)
    Delete right branch
```

Again test your application to ensure that you haven't broken anything.

6.17.8 Exercises

The following exercises build on the tree application developed. Some of these are tricky and will take time to complete.

1. There are two other types of order you can print the tree in - *pre-order* and *post-order*. Pre-order prints the nodes as *current node, left branch, right branch* and post-order prints the nodes as *left branch, right branch, current node.* Implement these two algorithms and test the application, comparing how the different orders work.

2. Implement a search function that returns true or false based on whether a value is in the tree.

3. Modify the tree node to use a `shared_ptr` rather than a raw pointer. What change is needed to the main application?

4. **For the Brave** - can you implement a copy operation to copy one tree to another? There are a number of ways this could be achieved, so try and work one out. In particular, pre-order and post-order should output the same order for the copied tree.

5. **For the Brave** - this one is tricky. Write an operation that allows the removal of a value from the tree. This will require you to reconfigure the tree by moving branches between nodes. There are four cases you need to deal with in your algorithm:

    (a) No node in the tree contains the data to be removed

    (b) The node containing the data has no branches

    (c) The node containing the data has one branch

    (d) The node containing the data has two branches

Unit 7

Debugging

We are now going to take a break from our examination of C++ as a programming language and look at one of the most important skills you can learn as a programmer - *debugging applications*. Testing is thought of as one of the most boring aspects of software development, but that is not really what we are doing here. We are looking at how we can examine our running program to discover the reasons why it isn't working properly.

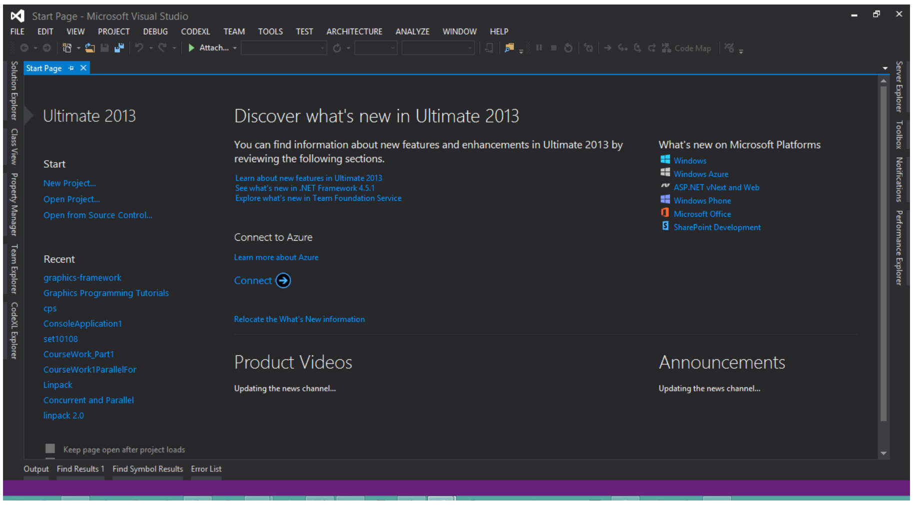
For this unit we are going to use Visual Studio. This is just how Microsoft provide their debugging tools. Other programming language compilers and environments provide other mechanisms for debugging and you should always find out these for the environment you are working in. We are going to cover the basic concepts of debugging in this unit. All these ideas should be available in any environment you are working in.

7.1 Getting Started with Visual Studio

To start off we need to get Visual Studio started. You will find Visual Studio under the following:

Applications ⇒ Faculty of Engineering, Computing, and Creative Industries ⇒ School of Computing ⇒ Visual Studio 2013 Run

Once Visual Studio has started (this may take a bit of time the first time) you should be presented with a screen similar to that shown in Figure 7.1.



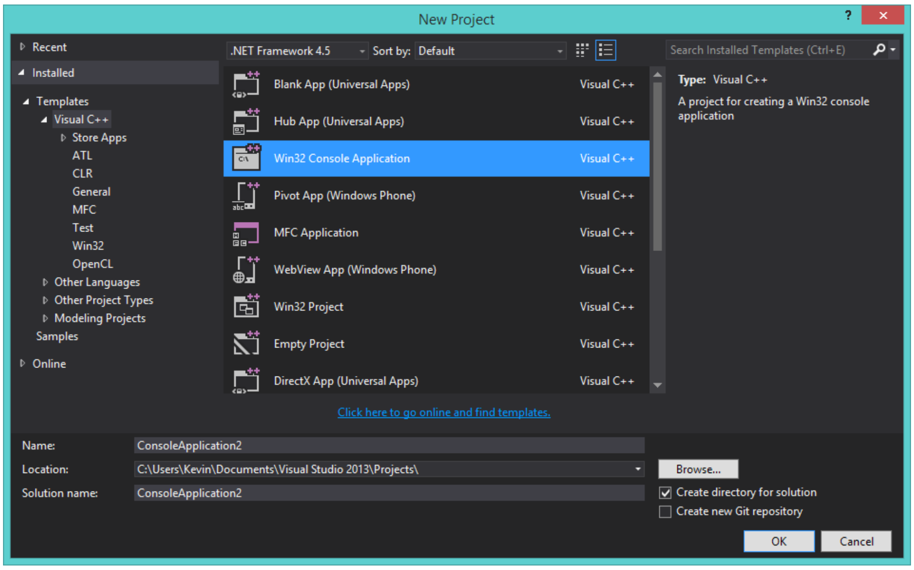
You can find information about new features and enhancements in Ultimate 2013 by reviewing the following sections.

Updating the news channel...

Updating the news channel...

Figure 7.1: Visual Studio 2013 Start Screen

Once Visual Studio has started up you need to create a new project. To do this select **File** ⇒ **New Project** in the menu. This will bring up the *New Project* window as shown in Figure 7.2.


New Project
Recent
Installed
Templates
Visual C++
Store Apps
ATL
CLR
General
MFC
Test
Win32
OpenCL
Other Languages
Other Project Types
Modeling Projects
Samples
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
Blank App (Universal Apps) Visual C++
Hub App (Universal Apps) Visual C++
Win32 Console Application Visual C++
Pivot App (Windows Phone) Visual C++
MFC Application Visual C++
WebView App (Windows Phone) Visual C++
Win32 Project Visual C++
Empty Project Visual C++
DirectX App (Universal Apps) Visual C++
Click here to go online and find templates.
Browse...
OK
Cancel

Type: Visual C++

A project for creating a Win32 console application

Name: ConsoleApplication2

Location: C:\Users\Kevin\Documents\Visual Studio 2013\Projects\

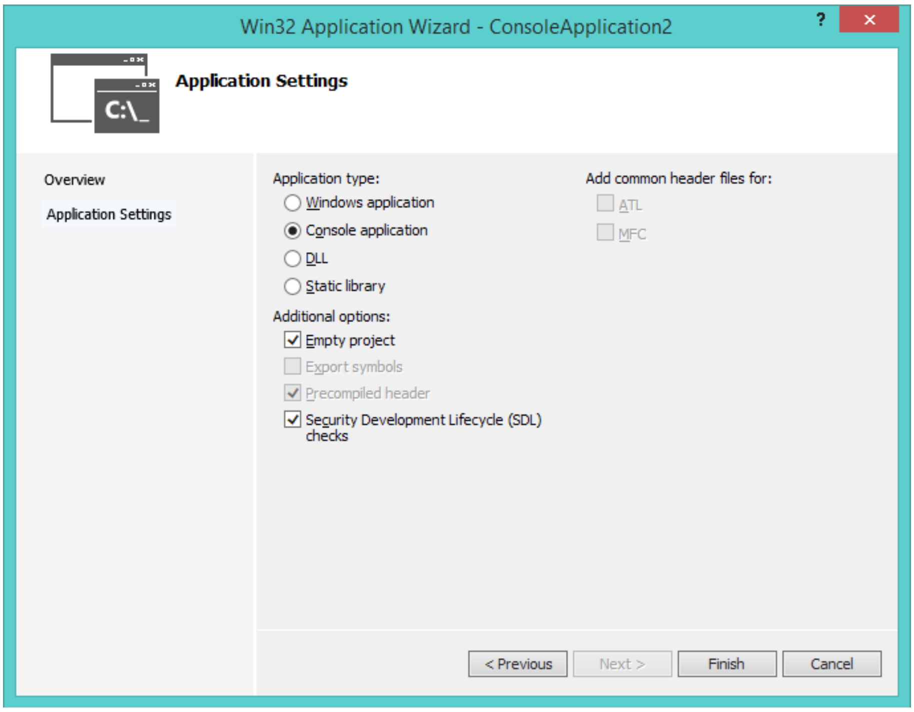
Solution name: ConsoleApplication2

☑ Create directory for solution

☐ Create new Git repository

Figure 7.2: New Project Window in Visual Studio

From here you need to find **New Win32 Console Application** under the **Visual C++** project templates type. This may be under **Other Languages** depending on what type of Visual Studio environment has been set as default. You can name your project whatever you want, although something sensible is a good idea (for example “Programming Fundamentals”). This will open up the *Visual Studio Project Configuration Window*. Make sure you select **Next** on this window to take you to the C++ project configuration options. This is shown in Figure 7.3.



Application type:

○ Windows application

● Console application

○ DLL

○ Static library

Additional options:

☑ Empty project

☐ Export symbols

☑ Precompiled header

☑ Security Development Lifecycle (SDL) checks

Add common header files for:

☐ ATL

☐ MFC

Figure 7.3: New Project Setup Configuration

Ensure that you have **Console Application** selected and also ensure that

Empty Project is selected. *You need to make sure that this is the case - if not your created project will be more complicated and you will have difficulty building it.*

Once the project is created your Visual Studio window should look similar to that shown in Figure 7.4. It may be the case that the *Solution Explorer* is on the right rather than the left shown in Figure 7.4.

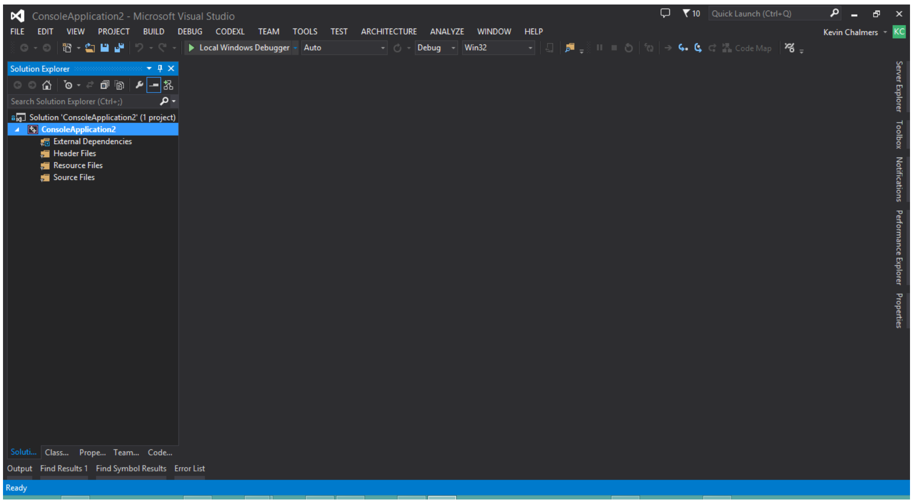

ConsoleApplication2 - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG CODEXL TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Quick Launch (Ctrl+Q)
Kevin Chalmers
Local Windows Debugger
Auto
Debug
Win32
Code Map
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ConsoleApplication2' (1 project)
ConsoleApplication2
External Dependencies
Header Files
Resource Files
Source Files
Server Explorer
Toolbox
Notifications
Performance Explorer
Properties
Soluti... Class... Prope... Team... Code...
Output Find Results 1 Find Symbol Results Error List
Ready

Figure 7.4: Visual Studio with Empty Project Created

Now that our project is created we need to add a code file which will contain our C++ code. To do this **Right click** on the **Project** in the **Solution Explorer**. From this menu select **Add** ⇒ **New Item** as shown in Figure 7.5.

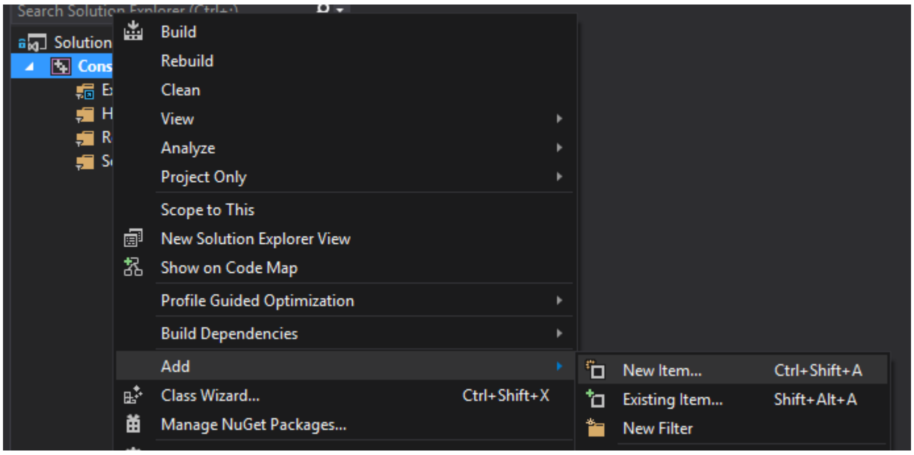


Figure 7.5: Adding a New File to the Visual Studio Project

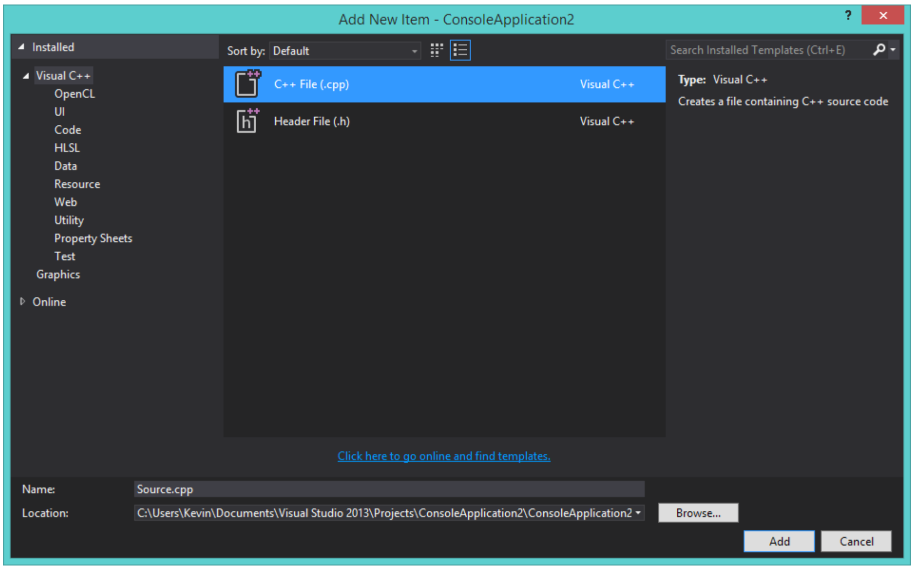
This will bring up the **New Item Window** shown in Figure 7.6.

Ensure that you have selected **C++ File (.cpp)** and give it a sensible name (although the default “Source.cpp” is fine). Click **Add** to return to the main window with an empty file.

For this first part of working with Visual Studio just use the basic “Hello World” application.

To run your application you can press the *green start button* illustrated in Figure 7.7. When you do this, your application will compile, link and run, opening (and quickly closing) a console window - blink and you will miss it.

And now you know how to create a project, add some code and run it in Visual Studio. We look into how you can stop the console window closing once the


Add New Item - ConsoleApplication2
Installed
Visual C++
OpenCL
UI
Code
HLSL
Data
Resource
Web
Utility
Property Sheets
Test
Graphics
Online
Search Installed Templates (Ctrl+E)
Click here to go online and find templates.
Browse...
Add
Cancel

Sort by: Default

C++ File (.cpp) Visual C++

Header File (.h) Visual C++

Type: Visual C++

Creates a file containing C++ source code

Name: Source.cpp

Location: C:\Users\Kevin\Documents\Visual Studio 2013\Projects\ConsoleApplication2\ConsoleApplication2

Figure 7.6: Visual Studio New File Window

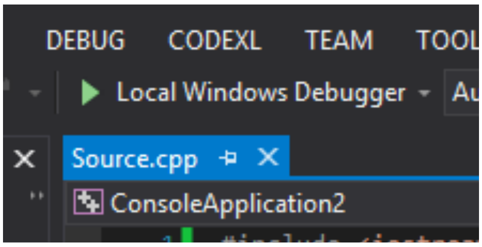


Figure 7.7: Debugging a Visual Studio Project

application exits shortly.

7.1.1 Where are my Files?

So we have been adding some C++ files but where are they stored? When you created the application you will have noticed that we had to select a location to create the project in. Your files are stored in this location. You can get to this folder quickly by **right clicking on the Project in the Solution Explorer** and selecting **Open Folder in File Explorer** as shown in Figure 7.8.

7.2 Starting without Debug

Pressing the start button in Visual Studio is in most cases the default behaviour when building an application, but it isn't necessary the best method when working with large projects. Visual Studio does allow you to compile individual files. With the file open in Visual Studio select **Build** ⇒ **Compile** in the menu to do this as shown in Figure 7.9.

To just build (not run) the application we select **Build** ⇒ **Build ¡Project Name¿** in the menu as shown in Figure 7.10.

To start our application so that the console window doesn't close we need to start the application without debugging. We do this by selecting **Debug** ⇒ **Start Without Debugging** from the menu as shown in Figure 7.11. Do this now to see the console window staying open after it has exited.

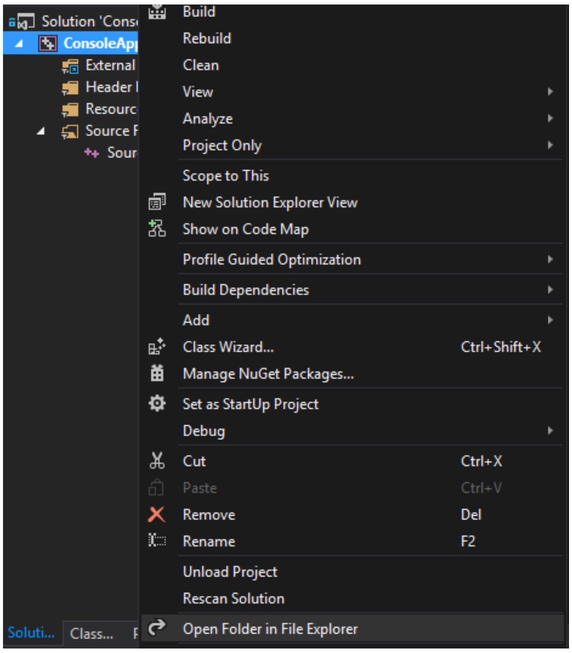


Figure 7.8: Opening Project File Location in Visual Studio

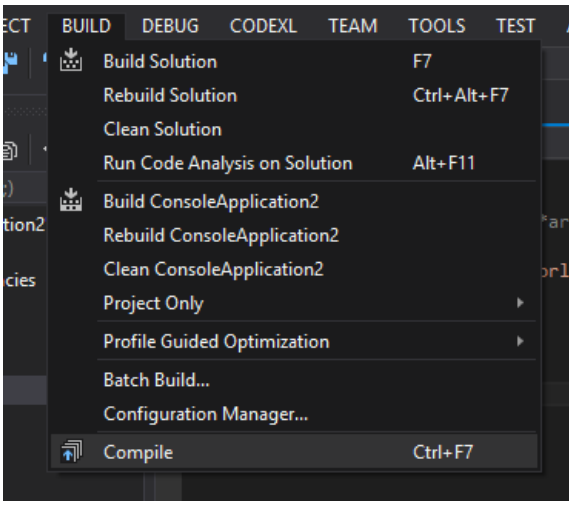

BUILD
DEBUG
CODEXL
TEAM
TOOLS
TEST
Build Solution F7
Rebuild Solution Ctrl+Alt+F7
Clean Solution
Run Code Analysis on Solution Alt+F11
Build ConsoleApplication2
Rebuild ConsoleApplication2
Clean ConsoleApplication2
Project Only
Profile Guided Optimization
Batch Build...
Configuration Manager...
Compile Ctrl+F7

Figure 7.9: Compiling a File in Visual Studio

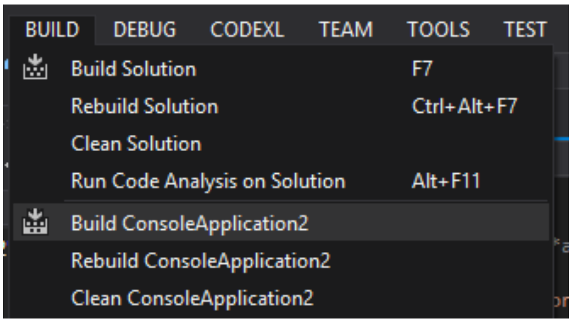

BUILD
DEBUG
CODEXL
TEAM
TOOLS
TEST
Build Solution F7
Rebuild Solution Ctrl+Alt+F7
Clean Solution
Run Code Analysis on Solution Alt+F11
Build ConsoleApplication2
Rebuild ConsoleApplication2
Clean ConsoleApplication2

Figure 7.10: Building a Project in Visual Studio

Compiling versus Building

We already covered the difference between compiling and building when we looked at compiling and linking back in the first unit. Compilation is the act of

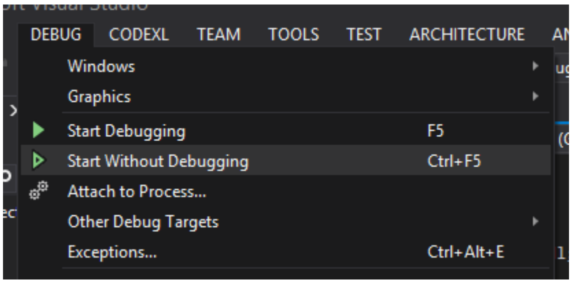


Figure 7.11: Running a Visual Studio Project without Debugging

converting a C or C++ code file to object code. Building is the act of linking the object code files and any added libraries to create an executable. Visual Studio just has options for you to control this.

7.3 What is a Project?

So in Visual Studio we create projects. But what is a project? In Visual Studio the projects are shown in the Solution Explorer as illustrated in Figure 7.12.

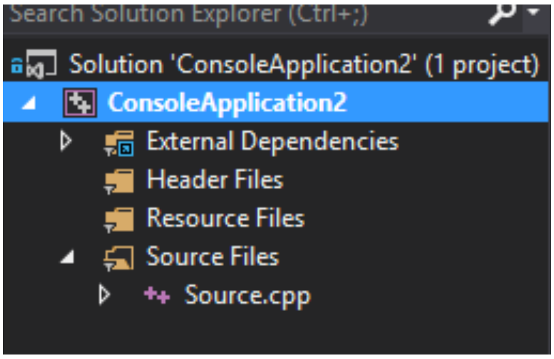


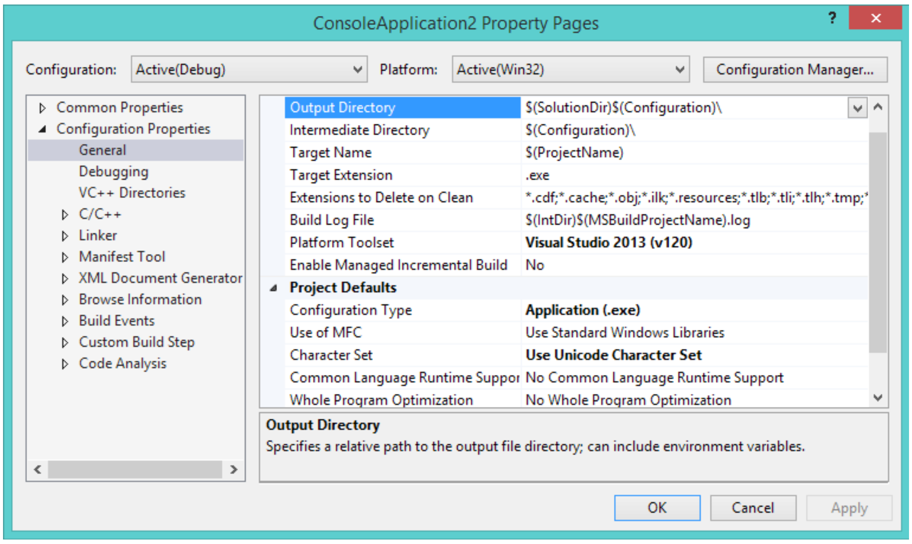
Figure 7.12: Visual Studio Project Highlighted in Solution Explorer

A project is just a way for Visual Studio to manage the build configuration of a collection of files. In effect, a project is just a make file underneath. We can set properties in Visual Studio and these change the build parameters used when calling the C/C++ compiler and linker. You can change these properties in the **Property Pages** window. This can be opened by **right clicking on the project** and selecting **Properties**. The Project Pages window is shown in Figure 7.13.

Projects themselves are contained in *Solutions*. Now let us add a new project to our solution.

7.4 Working with Multiple Project Solutions

If you **right click on the Solution in the Solution Explorer** and select **Add ⇒ New Project** (as shown in Figure 7.14) the *New Project Window* will open. Ensure that you set the correct project properties as before (Win32 Console Application and Empty Project). A new project will then be added to the solution.



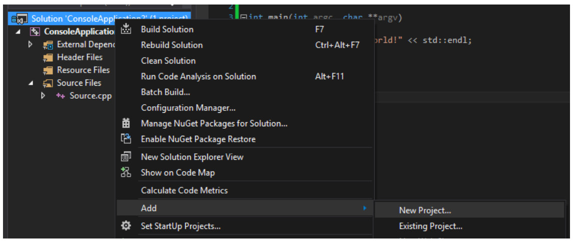
ConsoleApplication2 Property Pages

Configuration: Active(Debug)   Platform: Active(Win32)

Output Directory	$(SolutionDir)$(Configuration)\
Intermediate Directory	$(Configuration)\
Target Name	$(ProjectName)
Target Extension	.exe
Extensions to Delete on Clean	*.cdf;*.cache;*.obj;*.ilk;*.resources;*.tlb;*.tli;*.tlh;*.tmp;*
Build Log File	$(IntDir)$(MSBuildProjectName).log
Platform Toolset	**Visual Studio 2013 (v120)**
Enable Managed Incremental Build	No
Project Defaults	
Configuration Type	**Application (.exe)**
Use of MFC	Use Standard Windows Libraries
Character Set	**Use Unicode Character Set**
Common Language Runtime Suppor	No Common Language Runtime Support
Whole Program Optimization	No Whole Program Optimization

Output Directory
Specifies a relative path to the output file directory; can include environment variables.

Figure 7.13: Visual Studio Project Properties



```
int main(int argc, char **argv)
orld!" << std::endl;
```

Figure 7.14: Adding a Project to an Existing Solution in Visual Studio

This is going to be the project we want to run now. To do this, **Right click on the new project** and select **Set as StartUp Project** in the menu. This is shown in Figure 7.15.

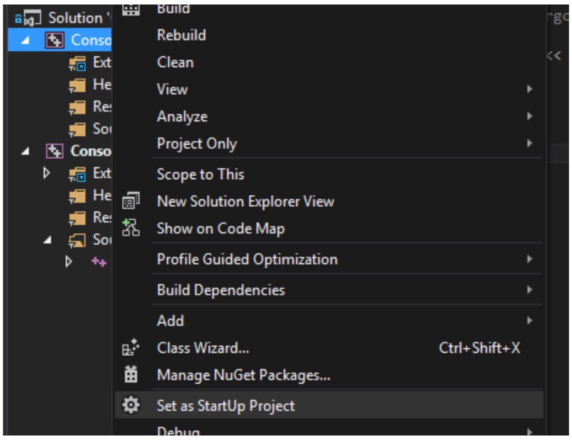


Figure 7.15: Setting the Startup Project in Visual Studio

For this project use the code for reading in user name using `cin` (Listing 5.6). We will start our work on debugging using this application.

7.5 Debug versus Release Builds

Visual Studio allows us to run an application in two different modes as standard - *Debug* and *Release*. These can be selected at the top of the Visual Studio window as shown in Figure 7.16.

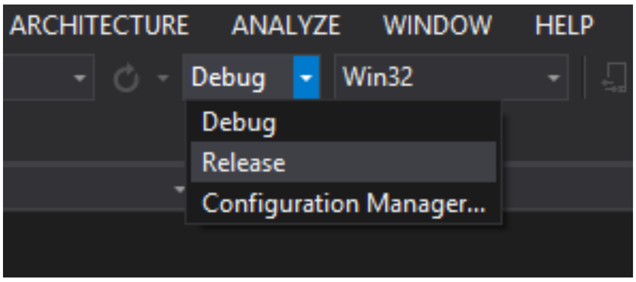


Figure 7.16: Changing Between Debug and Release Builds

A Debug application is the default. This type of application enables the Visual Studio debugger to attach to our application and thereby control the running of the application. This has a performance overhead but allows us to find bugs.

A Release application is meant to be the version you deliver to the client. It has no debug information added and is therefore better performing. The downside is that it doesn't have debug information.

Debug build also defines the `DEBUG` flag and Release the `RELEASE` flag. You did this yourself in the exercise in Section 4.1.4.

Ensure that your application is set as Debug for the rest of this unit. This is required for us to be able to debug our application.

7.6 Setting Breakpoints

We are now going to work with breakpoints. Breakpoints form the basis of debugging our applications. They allow us to stop our application running at a particular point. This is very handy in determining what state our application is in at when an error occurs.

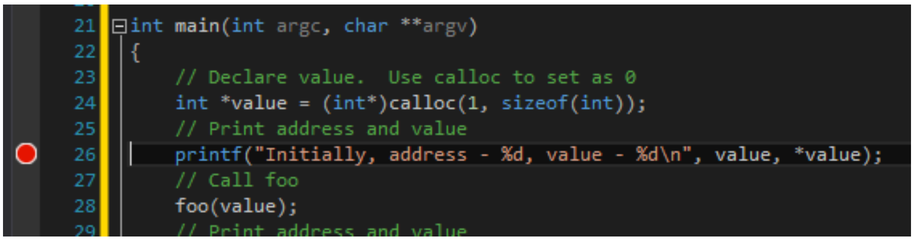
What is a Breakpoint?

In simple terms a breakpoint is a location in a program where the programmer wishes for the application to pause. When paused the programmer can examine properties of the running program such as the value of variables, the current call stack (more on this later) and even the registers.

When we encounter a bug in our code (either the application provides incorrect output or it fails) we put a breakpoint into our application in the approximate place where the application failed (this takes experience and knowledge of the program to determine). We then *step through* the program checking the values of variables and determining which line (or lines) of code was the culprit of the problem.

To say that debugging using breakpoints is one of the best skills to develop early as a programmer is an understatement. This is the go to method by any experienced programmer to try and find a bug in their code when developing software. You will find that any teacher will also immediately go to this technique when you have a problem with your code - it is just easier. This is why this part of the module is so important.

Depending on your environment there are different methods to set a breakpoint. Typically, in an *Integrated Development Environment* (IDE) like Visual Studio we do this by **left clicking** on the left of the line in the IDE. This is illustrated in Figure 7.17 where the red circle is the created breakpoint.



```
int main(int argc, char **argv)
{
    // Declare value.  Use calloc to set as 0
    int *value = (int*)calloc(1, sizeof(int));
    // Print address and value
    printf("Initially, address - %d, value - %d\n", value, *value);
    // Call foo
    foo(value);
    // Print address and value
```

Figure 7.17: Setting a Breakpoint in Visual Studio

For the new project you have just created (the one using `cin`) do this now on one of the lines (lines that are comments don't work). Then run the application by clicking the start button (*do not use Start Without Debugging*). Once your application hits this line it will pause. To continue the application just click the green start button (now called *Continue*) again. The application should then exited normally.

Try now with two breakpoints in the application. Then three. We can add as many breakpoints as we want. To remove a breakpoint just click on it. Remove all the breakpoints now.

In practice you should only set breakpoints in the area of code you think the problem lies. Once you are finished with a breakpoint you should remove it. If you don't, you might hit it the next time you run the application. This is more annoying than anything else and can be removed when hit. You should avoid leaving stray breakpoints around though.

7.7 Stepping Through a Program

Once an application is paused you can control how the debugger allows execution of the rest of the program. This is commonly called *stepping through the program*. This allows us to execute a line of code at a time and see the change in state this has caused.

We can actually start out application by stepping into it. This allows us to start the application by pausing on the first line. We do this by selecting **Debug** ⇒ **Step Into** in the menu as shown in Figure 7.18.

When stepping through the program we have three different techniques we can use:

Step Into - when the debugger is paused on a line that calls a function, step into that function.

Step Over - executes the current line that the debugger is paused on and moves onto the next line. If the line paused on is a function call, the function call is completely executed (*stepped over*).

Step Out - completes the currently executing function and returns to the calling function. We *step out of* the current function.

Figure 7.19 shows these buttons. These are will become visible when the application is being debugged. On the left there is the *Find Next Statement* button. This

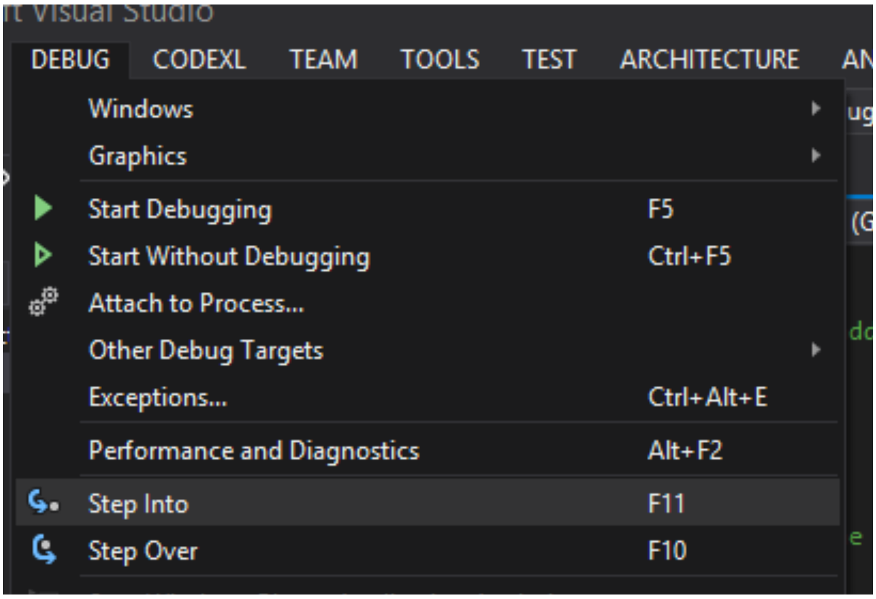


Figure 7.18: Stepping into a Program in Visual Studio

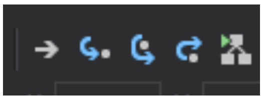


Figure 7.19: Debug Stepping Tools in Visual Studio

will take you to the current line paused on. Then we have *Step Into*, *Step Over*, and *Step Out*.

Step Into your application now and experiment with these buttons to see how the application executes. You can step into some of the functions called from the standard library, but don't expect to understand these at this point. Once the application is paused, you can hover the mouse over a variable to see its current value. This allows you to track the problems in your application.

7.8 assert

An often overlooked technique when building applications is the use of assertions. Assertions are a very powerful technique for ensuring that a function is called with correct parameters. We will look at how we do this shortly.

Assertions are so powerful that they can be used to form the basis of mathematically proving software correctness. These ideas are combined with concepts such as logic and set theory to allow this. The idea of proving software correctness is not covered in this module and falls under the area of *Formal Methods*.

What is an Assertion?

An assertion is a line of code we add to our application to make a statement about what *must be true* at that point in the execution of the application. It is just a function call with a logical condition in it as follows:

```
assert(condition);
```

The *condition* parameter can be any logical statement that can be equated to `true` or `false` such as `x <= 0` or `y == 0`.

If the condition in the call to `assert` is false then the application will stop with an error message. Otherwise it will continue as normal. This is the case when the application is being run in *Debug* mode. If in a *Release* build then the assertions are ignored. Therefore, we have the ability to perform extra checking during debugging but have this switched off when releasing our application.

The `assert` function comes from C and is part of the `cassert` header.

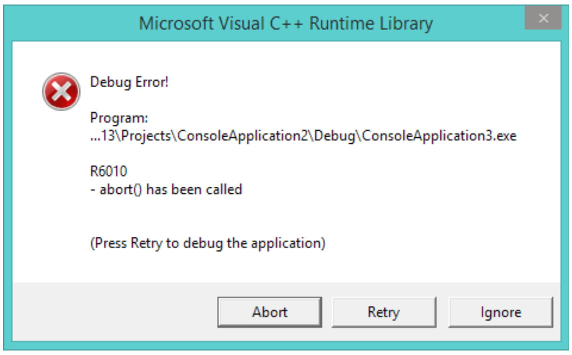
C++ also introduces *compile time* assertion checking through `static_assert`. This requires the use of values that are known at compile time and therefore is very restrictive. If a `static_assert` fails during compilation then the program will not compile. This is useful as it catches bug before even running the application. We won't go further into `static_assert` here however.

To test assertions, create a new project in Visual Studio as we did before and use the following code:

```
1  #include <iostream>
2  #include <cassert>
3
4  using namespace std;
5
6  float divide(float numerator, float denominator)
7  {
8      // Ensure that denominator does not equal 0
9      assert(denominator != 0);
10
11     // Return numerator / denominator
12     return numerator / denominator;
13 }
14
15 int main(int argc, char **argv)
16 {
17     // Call divide with 5 and 2
18     cout << divide(5.0f, 2.0f) << endl;
19
20     // Call divide with 6 and 1
21     cout << divide(6.0f, 1.0f) << endl;
22
23     // Call divide with 10 and 0
24     cout << divide(10.0f, 0.0f) << endl;
25
26     // We will get an error telling us line 9 is the problem during
           runtime
27
28     return 0;
29 }
```

Listing 7.1: Using an Assertion in C/C++

The use of `assert` is on line 9. When the assertion fails on this line (our call made on line 24) we get an error message telling us that line 9 is the problem. In Visual Studio a window is shown as illustrated in Figure 7.20.



Debug Error!

Program:
...13\Projects\ConsoleApplication2\Debug\ConsoleApplication3.exe

R6010
- abort() has been called

(Press Retry to debug the application)

Figure 7.20: Assertion Failure in Visual Studio

If run on the command line then the following similar message is output to the console:

```
Assertion failed: denominator != 0, file c:\users\kevin\documents\visual studio
2013\projects\consoleapplication2\consoleapplication3\source.cpp, line 9
```

Listing 7.2: Assertion Fail on the Command Line

Don't forget about assertions. They are so useful but programmers always seem to forget about them. You should always check the input to your functions and assertions are the smart way to do this.

Being Smart with Assertions

You should always write assertions for your functions. At the start of a function you should check to see if the parameters are in the accepted ranges. This is what we did in our sample application.

Another technique for using assertions placing them before calling a function to ensure that the state of the application is correct. This is especially useful when using external code.

Assertions also form the basis of the testing technique called *Unit Testing*. Again, this is beyond the scope of this module but you should come across it later in your studies.

7.9 Watching Variables

The point of debugging is to check variable values as the program is executing to ensure they are correct. We can make this easier by watching variables.

What do we mean by Variable Watching?

Variable watching means that we state we want to track the value of a variable. This is shown in the *Watch Window*. By default, Visual Studio will track local values by default, although it will try and be smart about it.

Variable watching is an important skill to develop alongside debugging. It is the whole point of what we are doing.

To test variable watching create a new project and use the following code.

```
#include <iostream>

```

```
using namespace std;

void outermost(int &value)
{
    // Multiply value by 2
    value *= 2;
}

void outer(int &value)
{
    // Add 5 to value
    value += 5;
    // Call outermost
    outermost(value);
}

void inner(int &value)
{
    // Divide value by 3
    value /= 3;
    // Call outer
    outer(value);
}

void innermost(int &value)
{
    // Subtract 4 from value
    value -= 4;
    // Call inner
    inner(value);
}

int main(int argc, char **argv)
{
    // Declare value - set to 100
    int v = 100;
    // Call innermost
    innermost(v);

    // Output value
    cout << "Value = " << 8v << endl;

    return 0;
}
```

Listing 7.3: Variable Watch Test Application

To add a variable watch in Visual Studio, `right click on the variable to watch` (in this case we will watch `v`) and then select **Add Watch**. Figure 7.21 illustrates.

Watched variables are shown in the *Watch Window* (you might need to open this window from **Debug** ⇒ **Windows** ⇒ **Watch** when the application is being debugged). This is shown in Figure 7.22.

Step through the application and examine the value of `v` through its running. Notice what happens when `v` is out of scope. Try watching the `value` variable and what happens with that. What does this tell you about variable watching? Also look at the *Autos* and *Locals* values as well.

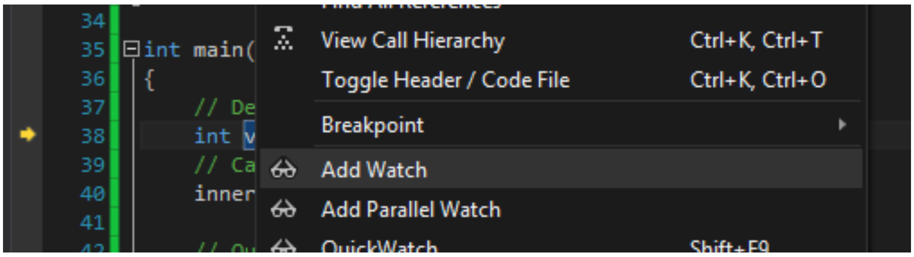
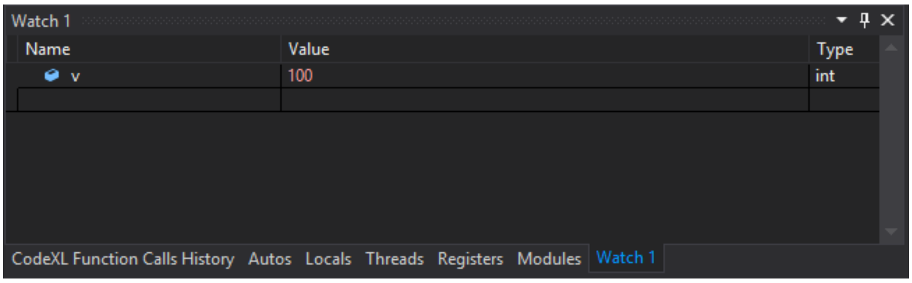


Figure 7.21: Watching a Variable in Visual Studio



Name	Value	Type
v	100	int

Figure 7.22: Visual Studio Watch Window

7.10 Advanced Breakpoints

Breakpoints can also have conditions attached to them. This allows us to determine under what circumstances the breakpoint should cause the application to pause.

When to Stop at a Breakpoint

We can use any boolean condition to determine when a breakpoint should pause. In Visual Studio we can also use the number of times the particular line of code is executed. This can be useful for loops.

Normally we can determine that a bug happens when a variable has a particular value, and therefore setting a conditional breakpoint can ensure we only pause when this condition is true.

Let us build a test application. The following application will cause our (x) to go beyond the limits storable in an int meaning that it wraps around. Let's capture when this happens.

```
#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
    // Starting value of x
    int x = 0;
    // Loop 100000 times
    for (int i = 0; i < 100000; ++i)
    {
        // Modify x
        x += 2;
        x *= 2;
    }

    // Output x
    cout << "x = " << x << endl;

    return 0;
}
```

Listing 7.4: Advanced Breakpoints Test Application

To set a conditional breakpoint, **right click on the breakpoint** and select **Condition**. This is shown in Figure 7.23.

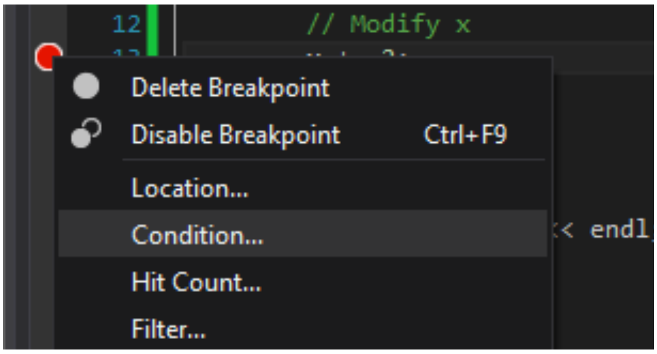
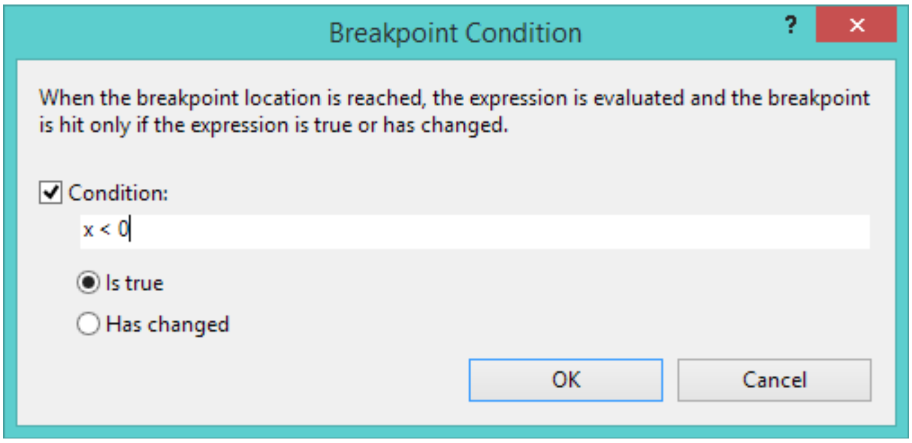


Figure 7.23: Setting an Advanced Breakpoint

This will open the *Breakpoint Condition Window* as shown in Figure 7.24. We will set the condition to `x < 0`.



When the breakpoint location is reached, the expression is evaluated and the breakpoint is hit only if the expression is true or has changed.

☑ Condition:

x < 0

● Is true

○ Has changed

Figure 7.24: Visual Studio Breakpoint Condition Window

Try running the application and capture when `x` wraps round. What is the value of `i` at this point?

7.11 Examining the Call Stack

Another useful feature of debugging is the call stack. For this section you should use the *Variable Watch* code (Listing 7.3).

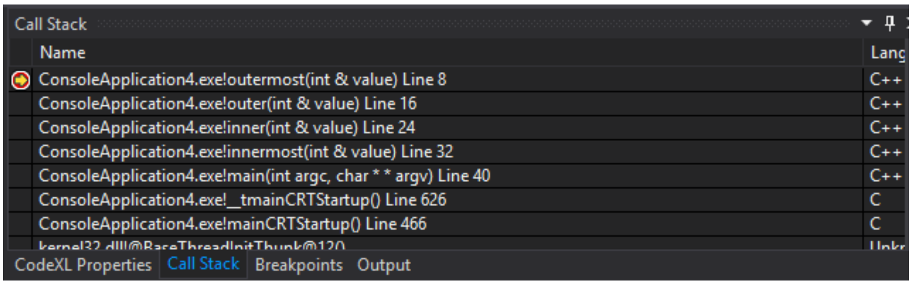
What is the Call Stack?

The call stack keeps track of the currently called functions. Each time we call a function, it is added to the call stack. When the function exits, the function is removed from the stack.

For our test application, our call stack looks as follows when we hit the `outermost` function:

outermost
outer
inner
innermost
main

The call stack in Visual Studio (Figure 7.25) allows us to click on one of the calling functions to see how the functions are called. This can have many uses when determining not just the state of variables but the order of execution.



Name	Lang
ConsoleApplication4.exe!outermost(int & value) Line 8	C++
ConsoleApplication4.exe!outer(int & value) Line 16	C++
ConsoleApplication4.exe!inner(int & value) Line 24	C++
ConsoleApplication4.exe!innermost(int & value) Line 32	C++
ConsoleApplication4.exe!main(int argc, char * * argv) Line 40	C++
ConsoleApplication4.exe!__tmainCRTStartup() Line 626	C
ConsoleApplication4.exe!mainCRTStartup() Line 466	C
kernel32.dll!@BaseThreadInitThunk@12()	Unkr

Figure 7.25: Visual Studio Call Stack

7.12 For the Brave - Using the Disassembly

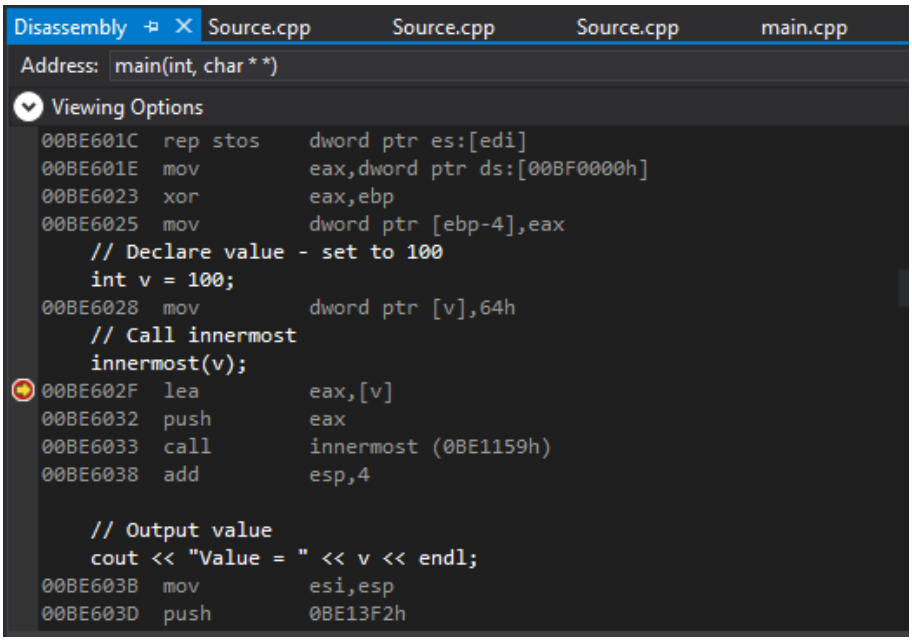
The next few sections are really C/C++ specific but you may find them useful. First of all, we can look at the assembly code being run in Visual Studio by looking at the *Disassembly* (**Debug** ⇒ **Windows** ⇒ **Disassembly**). Figure 7.26 shows what this looks like in Visual Studio. We can step through these instructions using the standard commands as well.

7.13 For the Brave - Examining the Registers

This low level of debugging allows us to interrogate individual machine instructions and see what is happening. The other part of this is to check what values the registers have. This can be done by opening the *Registers Window* (**Debug** ⇒ **Windows** ⇒ **Registers**). Figure 7.27 illustrates this view.

7.14 For the Brave - Examining Memory

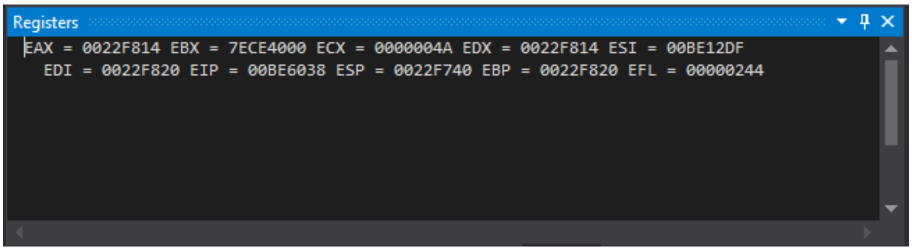
Visual Studio also allows you to examine raw memory (**Debug** ⇒ **Windows** ⇒ **Memory**). Figure 7.28 illustrates this view.



Address: main(int, char * *)

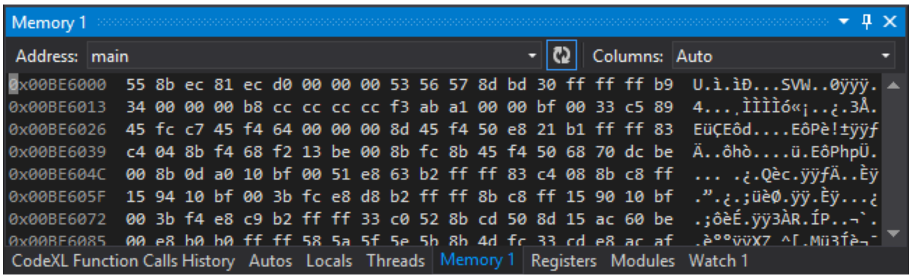
```
00BE601C  rep stos    dword ptr es:[edi]
00BE601E  mov         eax,dword ptr ds:[00BF0000h]
00BE6023  xor         eax,ebp
00BE6025  mov         dword ptr [ebp-4],eax
    // Declare value - set to 100
    int v = 100;
00BE6028  mov         dword ptr [v],64h
    // Call innermost
    innermost(v);
00BE602F  lea         eax,[v]
00BE6032  push        eax
00BE6033  call        innermost (0BE1159h)
00BE6038  add         esp,4

    // Output value
    cout << "Value = " << v << endl;
00BE603B  mov         esi,esp
00BE603D  push        0BE13F2h
```

Figure 7.26: Visual Studio Disassembly



```
EAX = 0022F814 EBX = 7ECE4000 ECX = 0000004A EDX = 0022F814 ESI = 00BE12DF
  EDI = 0022F820 EIP = 00BE6038 ESP = 0022F740 EBP = 0022F820 EFL = 00000244
```

Figure 7.27: Visual Studio Registers View



Address: main

Columns: Auto

```
0x00BE6000  55 8b ec 81 ec d0 00 00 00 53 56 57 8d bd 30 ff ff ff b9  U.ì.ìÐ...SVW..0ÿÿÿ.
0x00BE6013  34 00 00 00 b8 cc cc cc cc f3 ab a1 00 00 bf 00 33 c5 89  4...¸ÌÌÌÌó«¡..¿.3Å.
0x00BE6026  45 fc c7 45 f4 64 00 00 00 8d 45 f4 50 e8 21 b1 ff ff 83  EüÇEôd....EôPè!±ÿÿƒ
0x00BE6039  c4 04 8b f4 68 f2 13 be 00 8b fc 8b 45 f4 50 68 70 dc be  Ä..ôhò.¾..ü.EôPhpÜ.
0x00BE604C  00 8b 0d a0 10 bf 00 51 e8 63 b2 ff ff 83 c4 08 8b c8 ff  ... .¿.Qèc.ÿÿƒÄ..Èÿ
0x00BE605F  15 94 10 bf 00 3b fc e8 d8 b2 ff ff 8b c8 ff 15 90 10 bf  .”.¿.;üèØ.ÿÿ.Èÿ...¿
0x00BE6072  00 3b f4 e8 c9 b2 ff ff 33 c0 52 8b cd 50 8d 15 ac 60 be  .;ôèÉ.ÿÿ3ÀR.ÍP..¬`.
0x00BE6085  00 e8 b0 b0 ff ff 58 5a 5f 5e 5b 8b 4d fc 33 cd e8 ac af  .è°°ÿÿXZ_^[.Mü3Íè¬¯
```

Figure 7.28: Visual Studio Memory View

7.15 For the Brave - Detecting Memory Leaks

Windows can actually tell you if you have memory leaks in your applications. It does this by keeping track of how memory is allocated and freed. This is expensive, and should never be used in production code, but during the development cycle it can prove invaluable to remove these errors.

To enable this feature, the following two lines should be added at the very top of your main application file (before the other includes).

```
#define _CRTDBG_MAP_ALLOC
#include <crtdbg.h>
```

Listing 7.5: Enabling Leak Checking

These are part of the C runtime and will turn on the allocation monitoring. The other call you need to make should go as the first line of your main function. This allows automatic reporting.

```
_CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
```

Listing 7.6: Enable Automatic Reporting of Leaks

When your application exits in Visual Studio the output window should tell you if you have any leaks. You should write an application with a known leak to test this.

7.16 Exercises

You should experiment with the debugger at length, trying out some other applications. Getting a good familiarisation with the debugger will save you hours of work in the future. It is one of the best skills to work on as a new programmer.

For the Brave - look into `static_assert` and try and write an application using this construct. This can be used to fail a compilation so you don't even get to running your application. Write an application with `static_assert` that will cause the compilation to fail to test your understanding.

Unit 8

Object-Orientation and C++

We now return to our work on programming by moving onto object-orientation. Object-orientation is currently the most popular technique for software development in commercial software, and you will have already covered some of it's aspects. However, this module will delve a little further into object-orientation concepts that before, but not too far. The module Software Development 2 really goes into the detail here.

As a quick refresh of our journey through the module so far consider the following:

1. we looked at the basic idea of compiling and linking programs
2. we looked at how data is represented in memory and how we can define our own data types
3. we looked at how our code transforms in machine level instructions
4. we looked at how the pre-processor builds our compilation units and how we can include other code in our applications
5. we looked at how values are passed to our functions
6. we looked at how memory works and how we manage memory in our application taking into consideration the stack and heap
7. we looked at debugging

In this unit we will build on units 2 and 6 to further our ability to work with better data types and how these are represented in memory. We will also extend some of the ideas in unit 5 when we look at accessibility.

8.1 What is Object-Orientation?

Object-oriented programming revolves around the idea of an *object*. An object can be viewed simply as a collection of data (remember the `struct` data type we covered previously). These data items are known as the *attributes* (or fields or properties) of the object.

An object is more than just the data values that make it up however. An object also has *methods* associated with it (or functions, procedures). These methods are part of the object - they are not separate functions that can be called without an object.

Methods of an object can also access the attributes of the object as local variables. This is the general point of object-orientation - *the combination of data and function into a unit of abstraction*. Don't expect to become an object-oriented guru overnight. It will take a couple of years for the true power of the object abstraction to kick in. The idea of combining data and function is termed *encapsulation*.

8.2 Using struct in C++

Before diving into object-orientation proper, let us look back at struct and how we use it in C++. Remember that a struct is just a collection of data values that we combine together to form a new data type. This data type can be used as a standard variable. We can access the values in the struct using the dot notation, getting and setting values appropriately.

As C++ is an extension to C we can obviously use struct. However, C++ does make it a bit easier to use. The following application will illustrate.

```
#include <iostream>
#include <string>

using namespace std;

// Student struct in C++
// Note the use of string now rather than char*
struct student
{
  unsigned int matric;
  string name;
  string address;
};

// Prints student information
// Note we don't have to use 'struct student' now
// Also pass by reference
void print_student(student &s)
{
  cout << "matric no: " << s.matric << endl;
  cout << "name: " << s.name << endl;
  cout << "address: " << s.address << endl;
}

int main(int argc, char **argv)
{
  // Output sizeof student
  cout << "Size of student struct is " << sizeof(student) << "
      bytes" << endl;
  // Create a student using brackets construction
  student s =
  {
    42001290,
    "Kevin Chalmers",
    "School of Computing",
  };

  // Print student data
  print_student(s);

  return 0;
41 }
```

Listing 8.1: struct Usage in C++

First notice that we don't have to declare our variable as `struct student` on line 30 (or in the parameter pass on line 18). This reduces our typing overhead. We also pass the `student` by reference on line 18 to make our life easier.

What is Braces Initialisation?

Another new technique we have added in this lesson (this would also work in C) is *braces initialisation* on line 30. Here, we have actually defined the data values we want to have set for our new `struct`. This is effectively the same as defining these values in memory (the braces do that for us) and then stating that this is of type `student`.

Braces initialisation should generally be avoided in C++ as we can define *object constructors* to do this work for us. However, in C they can simplify our code.

Below is the output from this application.

```
Size of student struct is 52 bytes
matric no: 42001290
name: Kevin Chalmers
address: School of Computing
```

Listing 8.2: Output from C++ struct Application

Notice the size of this `struct` - 52 bytes. Where has this extra data come from? Remember that our data sizes are as follows:

- 4 bytes for our `unsigned int`
- 4 bytes for our `char*`
- 4 bytes for our `char*`

This is 12 bytes. Somehow C++ has added 40 bytes. Well this is the `string` object we have been using instead of `char*`. The `string` object in C++ contains more than just the `char*` - each `string` takes up 20 more bytes each. There are a number of values hidden behind the scenes, but one of them you should now - the `size` of the `string` (normally an `unsigned int` in 32-bit applications).

8.3 Data Sizes and Memory Representation of `struct`

We discussed the idea of memory representation of `struct`s in C previously. In C++ the same basic principles are also in place. We will be covering how objects are represented in memory based on their construction as we progress through the module.

We will now introduce some modelling to our work using the *Unified Modelling Language* (UML). We won't delve to far into UML here as this is covered in far more depth in your 2nd year of study. However, the diagrams here will aid your understanding. From UML we will use a *class diagram* to illustrate the application. For our test application the class diagram is given in Figure 8.1.

Each box represents one of our data types. We have the following:

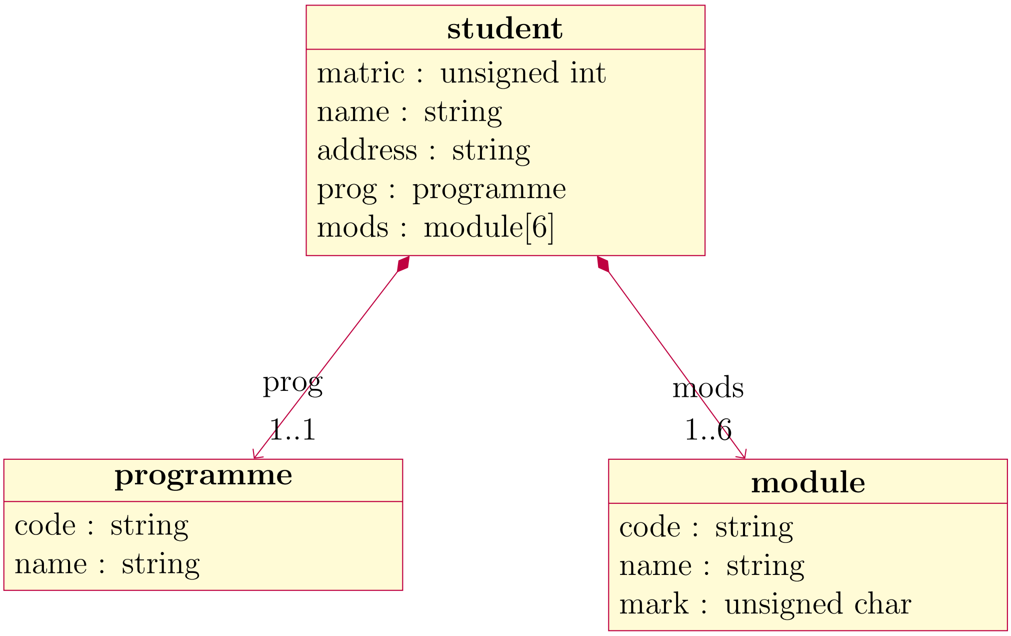


Figure 8.1: Class Diagram for Data Sizes Application

programme - a programme that a student can be enrolled on

module - a module a student can be enrolled on

student - a student

The attributes of the data types are given below the name. For `programme` we have a code of type `string` and name of type `string`. For `module` we have a code of type `string`, a name of type `string` and the mark of the module of type `unsigned char`. Finally, our `student` data type has a matric of type `unsigned int`, a name of type `string`, an address of type `string`, a prog of type `programme` and a mods of type `module[6]` (an array of 6 modules).

The other part of the diagram is the lines connecting the types. These tell us how our data types are *associated*. The connection between the `student` and other data types is called a *composition* relation - the `student` contains a programme and modules (there is more to it than that, but this will do for the moment). Our association has a direction (which we can communicate through the objects) and a multiplicity (1 to 1 or 1 to many).

From this class diagram alone you should be able to implement the basis for our test application. **You should try and implement the basic structure of the application yourself from the class diagram first**. The code below is the actual application. Look at this when you have made your own attempt.

```
#include <iostream>
#include <string>

using namespace std;

// Represents a programme of study
struct programme
{
  string code;
  string name;
};

```

```
13 // Represents a module studied
14 struct module
15 {
16   string code;
17   string name;
18   unsigned char mark;
19 };
20
21 // Represents a programme
22 struct student
23 {
24   unsigned int matric;
25   string name;
26   string address;
27   programme prog;
28   module mods[6];
29 };
30
31 int main(int argc, char **argv)
32 {
33   // Print the size of programme
34   cout << "Size of programme struct: " << sizeof(programme) << " bytes" << endl;
35   // Print the size of module
36   cout << "Size of module struct: " << sizeof(module) << " bytes" << endl;
37   // Print the size of student
38   cout << "Size of student struct: " << sizeof(student) << " bytes" << endl;
39
40   // ***** Add code to prompt the user for details and calculate year average *****
41
42   return 0;
43 }
```

Listing 8.3: Data Sizes of Combined Structures

Note on that the `student struct` declared on line 22 contains copies of our data types - not pointers or references. This means that our `student` object is 412 bytes in total (almost half a kilobyte). In memory, our `student` looks like the following:

Bytes	Data
0 - 3	`matric`
4 - 27	`name`
28 - 51	`address`
52 - 99	`prog`
100 - 151	`mods[0]`
152 - 203	`mods[1]`
204 - 255	`mods[2]`
256 - 307	`mods[3]`
308 - 359	`mods[4]`
360 - 411	`mods[5]`

8.3.1 Exercise

Complete the code for this application. Then expand it to allow details to be entered until the user enters end. An example output is shown below. As a hint, remember and use getline.

```
Size of programme struct: 48 bytes
Size of module struct: 52 bytes
Size of student struct: 412 bytes
Enter matric: 1234
Enter name: Kevin Chalmers
Enter address: Merchiston
Enter programme code: BH1234
Enter programme name: Software Engineering
Enter module 0 code: SET1234
Enter module 0 name: SD1
Enter module 0 mark: 85

... other modules

1234 Kevin Chalmers Software Engineering 78.3333
```

Listing 8.4: Output from C++ Data Size Application

8.4 Defining a class

Let us now move onto classes. A class in C++ is where we define what data (attributes) and operations (methods) our types have. It is an extension of our previous understanding of struct. We create instances of our classes which are called *objects*. This is the same idea as creating instances of our structs from before.

What is a class?
A class is the template for defining an object. It provides information on the values associated with the class (just like a struct) and the operations we can call on the object (the methods). There can also be further information to do with accessibility of data and operations and any inheritance that the class has (more on this later).

Below is our previous student application rewritten to use a class rather than a struct. Notice how similar they are. We will point out the differences through the rest of this unit.

```cpp
#include <iostream>
#include <string>

using namespace std;

// Define student as a class
class student
{
// We will come back to public soon
public:
  unsigned int matric;
  string name;
  string address;
};

// Print student details
void print_student(student &s)
{
  cout << "matric no: " << s.matric << endl;
20   cout << "name: " << s.name << endl;
21   cout << "address: " << s.address << endl;
22 }
23 
24 int main(int argc, char **argv)
25 {
26   // Output sizeof student
27   cout << "Size of student struct is " << sizeof(student) << "
       bytes" << endl;
28   // Create a student using brackets construction
29   student s =
30   {
31     42001290,
32     "Kevin Chalmers",
33     "School of Computing",
34   };
35 
36   // Print student data
37   print_student(s);
38 
39   return 0;
40 }
```

Listing 8.5: Defining a Class in C++

The first line to note is line 7. Here we use the keyword `class` rather than the keyword `struct`. Next we have line 9 where we use the keyword `public` followed by a colon. This denotes that everything after this point is *publicly accessible* outside the object. We will return to accessibility soon and its different types.

Our `class` definition then takes the same form as our `struct`. We have declared our three values as before. This definition will also be 52 bytes in size - defining it as a `class` makes no difference.

Notice again the use of braces initialization on line 29. A `class` behaves just as a `struct` in this regard as well.

The Difference Between a `struct` and a `class`

It seems that, apart from the use of `public`, a `class` and a `struct` behave almost identically. In fact, in C++ (not necessarily all languages), a `struct` and a `class` are indistinguishable. A `struct` can have methods and allows inheritance. At this point of our discussion the only difference is the use of the keyword `class` rather than `struct`.

The output for this application is given as follows:

```
Size of student struct is 52 bytes
matric no: 42001290
name: Kevin Chalmers
address: School of Computing
```

Listing 8.6: Output from `class` Application

8.5 Data Size of a `class`

Let us now return to our student record application that we implemented using `struct`s and replace these with `class` definitions. *Before using the code (below)* *you should attempt to implement the* **`class`** *based version yourself using the class diagram (Figure 8.1).*

```
#include <iostream>
#include <string>

using namespace std;

// Represents a programme of study
class programme
{
public:
  string code;
  string name;
};

// Represents a module studied
class module
{
public:
  string code;
  string name;
  unsigned char mark;
};

// Represents a programme
class student
{
public:
  unsigned int matric;
  string name;
  string address;
  programme prog;
  module mods[6];
};

int main(int argc, char **argv)
{
  // Print the size of programme
  cout << "Size of programme struct: " << sizeof(programme) << " bytes" << endl;
  // Print the size of module
  cout << "Size of module struct: " << sizeof(module) << " bytes" << endl;
  // Print the size of student
  cout << "Size of student struct: " << sizeof(student) << " bytes" << endl;

  // ***** Add code to prompt the user for details and calculate year average *****

  return 0;
}
```

Listing 8.7: Data Size of a Class

8.5.1 Exercise

Complete the application as you did previously with the `struct` version of the application. The output is the same as before.

8.6 public and private

OK, we skipped over the use of the keyword `public` in the last couple of examples, but this is an important aspect of object-orientation. Object-oriented programming involves the concept of *information hiding* (a form of encapsulation). This allows us to present a publicly facing interface to our object while hiding the inner workings of how these concepts are achieved.

Consider your smartphone. You happily use this device via its public interface but don't really have access to its hidden layer (the operating system) unless you jump through some hoops. Object-orientation is the same principle for programming. You can happily use an object (such as `string`) without getting into the details of how the object achieves its function.

Member Accessibility - `public` and `private`

In object-orientation we have a number of different accessibility types depending on the language we are using. The first two we will visit in C++ are the following:

public - can be accessed outside of the object

private - can only be accessed within the object (i.e. within the object's methods).

Accessibility becomes important when developing software to ensure correct usage of an object and its attributes. Although the accessibility can be broken (using techniques such as casting the object to raw memory then accessing the individual bytes) this is really going outside the intended purpose of the object. You should adhere to accessibility to make your programming easier. Hiding unnecessary details and working with an object on its interface makes life easier. You wouldn't want to solder individual chips onto your smartphone to get more storage would you? A SD card's public interface does this job for you.

Let us now build an example application using `public` and `private` member accessibility and see what the compiler does when we try and access `private` members outside the object.

```cpp
#include <iostream>

using namespace std;

// Struct with public and private members
struct my_struct
{
  // Default access of struct is public
  unsigned int x = 10;
private:
  // This value is not accessible outside my_struct
  unsigned int y = 20;
};

class my_class
{
  // Default access of class is private
  unsigned int x = 10;
19 public:
20   // This value is accessible outside my_class
21   unsigned int y = 20;
22 };
23
24 int main(int argc, char **argv)
25 {
26   // Declare objects
27   my_struct a;
28   my_class b;
29
30   // This line will compile
31   cout << "a.x = " << a.x << endl;
32   // This line will not compile
33   cout << "a.y = " << a.y << endl;
34   // This line will not compile
35   cout << "b.x = " << b.x << endl;
36   // This line will compile
37   cout << "b.y = " << b.y << endl;
38
39   // Try and set the values
40   a.x = 20;
41   a.y = 40;
42   b.x = 30;
43   b.y = 60;
44
45   // This line will compile
46   cout << "a.x = " << a.x << endl;
47   // This line will not compile
48   cout << "a.y = " << a.y << endl;
49   // This line will not compile
50   cout << "b.x = " << b.x << endl;
51   // This line will compile
52   cout << "b.y = " << b.y << endl;
53
54   return 0;
55 }
```

Listing 8.8: Declaring `public` and `private` Object Members

The compiler will output the following errors:

```
C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\INCLUDE\xlocale(337) : wa
rning C4530: C++ exception handler used, but unwind semantics are not enabled. S
pecify /EHsc
public_private.cpp(33) : error C2248: 'my_struct::y' : cannot access private mem
ber declared in class 'my_struct'
        public_private.cpp(12) : see declaration of 'my_struct::y'
        public_private.cpp(7) : see declaration of 'my_struct'
public_private.cpp(35) : error C2248: 'my_class::x' : cannot access private memb
er declared in class 'my_class'
        public_private.cpp(18) : see declaration of 'my_class::x'
        public_private.cpp(16) : see declaration of 'my_class'
public_private.cpp(41) : error C2248: 'my_struct::y' : cannot access private mem
ber declared in class 'my_struct'
        public_private.cpp(12) : see declaration of 'my_struct::y'
        public_private.cpp(7) : see declaration of 'my_struct'
public_private.cpp(42) : error C2248: 'my_class::x' : cannot access private memb
er declared in class 'my_class'
        public_private.cpp(18) : see declaration of 'my_class::x'
        public_private.cpp(16) : see declaration of 'my_class'
public_private.cpp(48) : error C2248: 'my_struct::y' : cannot access private mem
ber declared in class 'my_struct'
        public_private.cpp(12) : see declaration of 'my_struct::y'
        public_private.cpp(7) : see declaration of 'my_struct'
public_private.cpp(50) : error C2248: 'my_class::x' : cannot access private memb
er declared in class 'my_class'
```

```
        public_private.cpp(18) : see declaration of 'my_class::x'
        public_private.cpp(16) : see declaration of 'my_class'
```

Listing 8.9: Compiler Errors for Accessing `private` Members

Take a look at these errors and recognise them. It is likely you will hit these a number of times during your software development education. Each error tells you the line the problem occurs (e.g. line 33), the line where the definition is restricting access (e.g. line 12), and where the definition of the data type begins (e.g. line 7).

Also note that we are defining both `struct` and `class` data types here and can freely use `public` and `private` in either. However, there is a subtle difference between `struct` and `class` now which we will define.

The Difference Between a `struct` and a `class` (Revisited)

OK we previously said that `struct` and `class` had no difference in C++. This is true except for one aspect - *default accessibility*. In C++, members of a `struct` are `public` by default whereas in a `class` they are `private` by default. That's it. No other differences exist. However, it is normally considered good practice to treat `class` as data types with object-orientation concepts (methods, inheritance, etc.) and `struct` as just a basic data type with publicly accessible data - in effect a record.

8.7 Defining Methods

Now that we have covered the basic idea of defining a class in C++ (it is just a `struct` with a different keyword and different default accessibility) let us move onto adding the new concept - methods. Remember that a method is just a function associated with an object rather than independent of it. Therefore it considers the object's attributes in its scope.

What do we Mean by a Method?

A method is a function associated with an object. This means that the defined function has complete access to the object's properties. Whereas before you would have passed the data type into the function as follows:

```
do_something(a, b, c);
```

We can now just call `do_something` on the object itself:

```
a.do_something(b, c);
```

The method has access to `a`'s attributes within it's local scope. The method can also access any `private` values.

Declaring a Method

This is obviously language dependant, but the general idea of defining a method for a class is to add it to the class definition. For example, in C++ we could write the following:

```
class my_class
{
private:
    int x;
    int y;
public:
    int add()
    {
        return x + y;
    }
};
```

In most programming languages this is the general style of defining methods - inline with the class definition.

In C++ we can also declare the method inline but define it elsewhere in our code base. We do this as follows:

```
class my_class
{
private:
    int x;
    int y;
public:
    int add();
};

// Other code, maybe even a separate file

int my_class::add()
{
    return x + y;
}
```

Best practice in C++ normally states that we should declare our classes in header files and then implement the actual functions (their definitions) in separate code files. This means that when we share our code we further hide the internal workings. However, C++ relies heavily on templates in certain areas, and these don't allow this general idea. Therefore, there is no hard and fast rule for defining methods inline in classes.

One piece of advice though is that you should try and keep your `class` definitions small. Therefore, if your method has more than one or two lines of code you should declare it in the class and then define it outside of the class. This will make your code easier to manage in the long run.

Let us now look at an example application using methods. This is an extension of our last application, but this time we are overcoming the accessibility problems by utilising accessor methods. The code is below.

```
#include <iostream>

using namespace std;

struct my_struct
{
  // Publicly accessible
  unsigned int x = 10;
private:
  // Privately accessible
  unsigned int y = 20;

public:
  // Public methods
  unsigned int get_y()
  {
    return y;
  }

  void set_y(unsigned int value)
  {
    y = value;
  }
};

class my_class
{
  // Privately accessible
  unsigned int x = 10;
public:
  // Publicly accessible
  unsigned int y = 20;

  // Public methods
  unsigned int get_x()
  {
    return x;
  }

  void set_x(unsigned int value)
  {
    x = value;
  }
};

int main(int argc, char **argv)
{
  // Declare objects
  my_struct a;
  my_class b;

  // Can access a.x directly
  cout << "a.x = " << a.x << endl;
  // Use get_y on a to access a.y
  cout << "a.y = " << a.get_y() << endl;
  // Use get_x on b to access b.x
  cout << "b.x = " << b.get_x() << endl;
  // This line will compile
  cout << "b.y = " << b.y << endl;

  // Try and set the values
  a.x = 20;
  a.set_y(40);
  b.set_x(30);
  b.y = 60;

  // Can access a.x directly
  cout << "a.x = " << a.x << endl;
  // Use get_y on a to access a.y
```

```
    cout << "a.y = " << a.get_y() << endl;
    // Use get_x on b to access b.x
    cout << "b.x = " << b.get_x() << endl;
    // This line will compile
    cout << "b.y = " << b.y << endl;

    return 0;
}
```

Listing 8.10: Defining Methods

You should recognise the general concept of being able to call methods on the objects created. The output from this object is as follows:

```
a.x = 10
a.y = 20
b.x = 10
b.y = 20
a.x = 20
a.y = 40
b.x = 30
b.y = 60
```

Listing 8.11: Output from Methods Application

8.8 Object Construction

Up until now we haven't really discussed what happens when we create objects in our C++ applications. This is a process called *object construction*. Obviously we have been creating objects (instances of classes) in our code already, but now we will now look at what happens when objects are created and how we control this using an object *constructor*.

Writing a Constructor

A constructor is just a special type of method defined for a `class`. Each `class` has a default constructor if no other constructor is defined. This is used to define the initial values of the object.

A constructor is a method with no return type using the name of the class as its name. A default constructor for a `class` can therefore be defined by the programmer as follows in C++:

```
class my_class
{
private:
    int x;
    int y;
public:
    my_class()
    {
        x = 0;
        y = 0;
    }
};
```

We can now create an object by specifically calling this constructor as follows:

`my_class x = my_class();` or

`my_class x();`

We can also define a constructor that takes parameters. Let us extend the above value as follows:

```
class my_class
{
private:
    int x;
    int y;
public:
    my_class()
    {
        x = 0;
        y = 0;
    }
    my_class(int a, int b)
    {
        x = a;
        y = b;
    }
};
```

Our second constructor allows us to explicitly set the values of the attributes. In this case we can call the constructor as follows:

`my_class x = my_class(5, 10);` or
`my_class x(5, 10);`

Normally we want to write our own constructors so that we can construct internal objects in a particular way and also so that we can set the initial attributes.

The following application shows how we can write constructors for both `struct` and `class` data types.

```
#include <iostream>

using namespace std;

struct my_struct
{
  unsigned int x;
  unsigned int y;

  my_struct(unsigned int a, unsigned int b)
  {
    x = a;
    y = b;
  }
};

class my_class
{
public:
  unsigned int x;
  unsigned int y;

  my_class(unsigned int a, unsigned int b)
  {
    x = a;
    y = b;
  }
};

int main(int argc, char **argv)
{
  // Create objects
  my_struct a(10, 20);
  my_class b(10, 20);

  // Print details
  cout << "a.x = " << a.x << endl;
  cout << "a.y = " << a.y << endl;
  cout << "b.x = " << b.x << endl;
  cout << "b.y = " << b.y << endl;

  // Reallocate a and b
  a = my_struct(20, 40);
  b = my_class(30, 60);

  // Print details
  cout << "a.x = " << a.x << endl;
  cout << "a.y = " << a.y << endl;
  cout << "b.x = " << b.x << endl;
  cout << "b.y = " << b.y << endl;

  return 0;
}
```

Listing 8.12: Using Object Constructors

The output from this application will be the same as the last one. The point of this application is to introduce the idea of object construction and what happens. So far we have covered how objects are represented in memory and how they are constructed. Let us now combine these ideas together and the implications of object construction order.

8.9 Object Construction Order

Understanding how our objects is important. The order our objects are created has knock on effects to the order of instructions that are executed. Realising that certain values are initialised in a certain order is fundamental as it dictates the order of instructions that are executed and the order in which memory is allocated.

Objects are generally constructed in the following manner:

1. The memory for the object is allocated. This is the total memory requirements

2. The attributes of the object are initialised in the order that they are declared either by calling their default constructor or by the programmer calling a particular constructor. These attributes in turn call any constructors for their internal objects.

3. The body of the constructor of the object being created is called.

So in our previous student record example the following should occur:

1. The memory for the student is allocated.
2. The constructor for the name and address attributes are called (both `string`).
3. The constructor for the programme is called. This calls the constructor for `string` twice.
4. The constructor for each of the modules is called. This also calls the constructor for `string` twice.
5. The body of the student constructor is executed, completing the object construction.

So to create our `student` object a total of 24 function calls are made:

- A call to the `student` constructor
- Two calls to the `string` constructor
- A call to the `programme` constructor
- Two calls to the `string` constructor
- For each module (six) a call to the `module` constructor
- For each module (six) a call to two `string` constructors.

That is a lot of calls just to create our `student` object.

Object Construction Order

Realising the order of calls made when constructing an object is crucial for understanding when values have been initialised. It may be the case that you wish to use some of your data in an object to initialise its internal objects. Doing this in the wrong order will lead to errors with uninitialised data and other issues.

Just as a reminder an object is constructed in the following order:

1. The memory for the object is allocated. This is the total memory requirements
2. The attributes of the object are initialised in the order that they are declared either by calling their default constructor or by the programmer calling a particular constructor. These attributes in turn call any constructors for their internal objects.
3. The body of the constructor of the object being created is called.

Why are we Interested in Object Construction Order?

We will be building more complex objects through the rest of the module and we will be adding to our object construction order code. Inheritance also plays a role in object construction, and using pointers means that we can forgo actual object construction until later.

However, object construction order does allow us to realise something - *a class cannot define attributes which are references.* Remember that a reference must point to a created object (it cannot be null). It is of itself not a value but a reference to one. Therefore you cannot "create" a reference and we must assign a value to the reference when it is created. There are ways around this, but if you need to "point" to another data type in a C++ class it is wisest to use a pointer (or better yet a smart pointer).

To illustrate object construction order consider the following application. You will need to complete it as before. It is an extension of our student mark calculator but using defined constructors.

```cpp
#include <iostream>
#include <string>

using namespace std;

// Defines a programme of study
struct programme
{
  string code;
  string name;

  // Default constructor
  programme()
  {
    cout << "Called programme default constructor" << endl;
  }

  // Constructor with parameters
  programme(const string &code, const string &name)
  {
    cout << "Called programme parametrised constructor" << endl;
    this->code = code;
    this->name = name;
  }
};

// Defines a module
struct module
{
  string code;
  string name;
  unsigned char mark;

  // Default constructor
  module()
  {
    cout << "Called module default constructor" << endl;
  }

  // Constructor with parameters
  module(const string &code, const string &name, unsigned int mark)
  {
    cout << "Called module parametrised constructor" << endl;
    this->code = code;
    this->name = name;
    this->mark = mark;
  }
};

// Defines a student
struct student
{
  unsigned int matric;
  string name;
  string address;
  programme prog;
  module mods[6];

  // Default constructor
  student()
  {
    cout << "Called student default constructor" << endl;
  }
};

int main(int argc, char **argv)
{
  cout << "Creating a student object" << endl;
  student s;

  // Set some student values
  cout << "Setting student details" << endl;
  s.matric = 42001290;
  s.name = "Kevin Chalmers";
  s.address = "School of Computing";

  // Set the programme
  cout << "Setting programme" << endl;
  s.prog = programme("56119BH", "BEng (Hons) Software Engineering")
     ;

  // Set the modules
  cout << "Setting module 0" << endl;
  s.mods[0] = module("SET07109", "Programming Fundamentals", 85);

  // **** Add other modules ****

  // **** Calculate average mark ****

  // **** Print average mark ****

  return 0;
}
```

Listing 8.13: Object Construction Order

Default Constructor

Notice that as well as creating a parametrised constructor we must also define the default one. In C++, a default constructor is only defined if no other constructor is defined. As soon as you define a constructor for a class if you want a default constructor you must define it.

In this application case it does provide an illustration of the method calls invoked during object construction. In general you should only provide a default constructor if you absolutely need one.

The `this` Pointer

We have introduced another new keyword in this application - `this`. The `this` pointer refers to the current object that a method is being called on. It exists as a attribute within the scope of the object. It allows us to access the individual attributes and methods of the objects and highlight that we are specifically calling these ones.

In practice C++ programmers don't often use `this`. It can be useful to aid understanding of your code, or to avoid conflicts (such as our example where the parameters passed to the constructor have the same names as our attributes). It can be useful to use when you cannot remember the names of your attributes or methods in a class as well.

Pointer to Operator

The `this` pointer also uses a new notation - `->`. We will cover this shortly. Effectively this is used to access attributes and methods from a pointer rather than dereferencing the pointer first. It is just short hand to make code a little clearer. We will cover this idea in more depth later.

Running the application will give you an output as follows:

```
Creating a student object
Called programme default constructor
Called module default constructor
Called module default constructor
Called module default constructor
Called module default constructor
Called module default constructor
Called module default constructor
Called student default constructor
Setting student details
Setting programme
Called programme parametrised constructor
Setting module 0
Called module parametrised constructor

... other module construction
```

Listing 8.14: Output from Object Construction Application

Notice the order of constructors called. This is the order that we said the object would be constructed. However of note is what happens when we assign values to the `student` object. This calls the relevant constructor of either `programme` or `struct`. So although our application did allocate initialise the values, a new constructor was called when we set specific values.

8.9.1 Exercise 1

As in our previous similar applications, you need to complete this application. The point here is to recognise the order that the objects are created in. Understanding the order of operations in an application is very important as it can change your understanding of how the application runs.

8.9.2 For the Brave - Exercise 2

To avoid these unnecessary constructor calls for our object we can define how our internal objects are constructed by invoking their parametrised constructors in our own constructor. The following code sample illustrates this idea.

```
class A
{
private:
    string str;
public:
    A(const string &s)
    : str(s) // Calls the string constructor
    {
    }
};

class B
{
private:
    string str;
    A a;
public:
    B(const string &s)
    : str(s), a("Hello") // Calls string and A constructor
    {
    }
};
```

Notice the use of a colon followed by the calls to the relevant constructors. Rewrite the student record application to use these calls to ensure that the `string` values are created correctly and that then that a student object can be fully initialised in one call (warning - this will require a lot of parameters. This is *very bad* practice).

8.10 Object Destruction

Object construction is what occurs when we create a new object (this includes copying the object to a function using pass by value). Object destruction is what happens when an object is destroyed. Whereas you may have come across the idea of object construction, object destruction is not normally taught in languages such as Java and C# because of the garbage collector. In C++ they are essential as we may have to free up memory that the object has created.

Object destruction occurs whenever an object is deleted. This could happen when it goes out of scope (stack based values) or when `delete` is called on an object previously created on the heap. Object destruction calls a *destructor* on the object. Be warned - you should never call a destructor yourself. It will be called for you when the object is deleted.

Object destruction is necessary for managing resources in C++. It is used extensively in smart pointers (where the constructor increments the use count and the destructor decrements it). Although you probably won't see much use in object destructors at this time, when you start building large applications they become fundamental.

Declaring a Destructor

In C++ (and C# for that matter) an object destructor looks like a constructor except that it has a tilde ( ) in front of it. It cannot take any parameters since we do not call it - it is automatically called. Therefore a typical destructor would look as follows:

```cpp
class my_class
{
    // other declarations
public:
    ~my_class()
    {
        // clean up resources
    }
};
```

Notice that the destructor is declared `public`. **A destructor must always be public so it can be called when the object is deleted.** Otherwise, a destructor should effectively undo what the constructor did (and undo any resource allocation undertaken during the object's lifetime).

Let us now modify our student record application so that it includes destructors. This is shown below.

```cpp
#include <iostream>
#include <string>

using namespace std;

// Defines a programme of study
struct programme
{
  string code;
  string name;

  // Default constructor
  programme()
  {
    cout << "Called programme default constructor" << endl;
  }

  // Constructor with parameters
  programme(const string &code, const string &name)
  {
    cout << "Called programme parametrised constructor" << endl;
    this->code = code;
    this->name = name;
  }

  // Destructor for programme
  ~programme()
  {
    cout << "Called programme destructor" << endl;
  }
};

// Defines a module
struct module
```

```cpp
{
  string code;
  string name;
  unsigned char mark;

  // Default constructor
  module()
  {
    cout << "Called module default constructor" << endl;
  }

  // Constructor with parameters
  module(const string &code, const string &name, unsigned int mark)
  {
    cout << "Called module parametrised constructor" << endl;
    this->code = code;
    this->name = name;
    this->mark = mark;
  }

  // Module destructor
  ~module()
  {
    cout << "Called module destructor" << endl;
  }
};

// Defines a student
struct student
{
  unsigned int matric;
  string name;
  string address;
  programme prog;
  module mods[6];

  // Default constructor
  student()
  {
    cout << "Called student default constructor" << endl;
  }

  // Student destructor
  ~student()
  {
    cout << "Called student destructor" << endl;
  }
};

int main(int argc, char **argv)
{
  cout << "Creating a student object" << endl;
  student s;

  // Set some student values
  cout << "Setting student details" << endl;
  s.matric = 42001290;
  s.name = "Kevin Chalmers";
  s.address = "School of Computing";

  // Set the programme
  cout << "Setting programme" << endl;
  s.prog = programme("56119BH", "BEng (Hons) Software Engineering")
    ;

  // Set the modules
  cout << "Setting module 0" << endl;
  s.mods[0] = module("SET07109", "Programming Fundamentals", 85);

  // **** Add other modules ****

  // **** Calculate average mark ****

  // **** Print average mark ****

  return 0;
}
```

Listing 8.15: Object Destruction

Destruction Order

Destruction happens almost in reverse to construction. The following order of calls are made:

1. The destructor of the object being deleted is invoked.
2. The destructors of the attributes of the object are invoked in reverse order to how the attributes are declared.
3. The memory of the deleted object is freed

So for our student record application we again invoke 24 function calls to delete a `student`:

1. A call to the `student` destructor
2. Six calls to `module` destructor
3. Each `module` destructor calls two `string` destructors
4. A call to the `programme` destructor
5. The `programme` invokes two `string` destructors
6. Two `string` destructors are invoked

Again a lot of calls are made to destroy our simple `student` object.

Running this application will provide the following output:

```
Creating a student object
Called programme default constructor
Called module default constructor
Called module default constructor
Called module default constructor
Called module default constructor
Called module default constructor
Called module default constructor
Called student default constructor
```

```
Setting student details
Setting programme
Called programme parametrised constructor
Called programme destructor
Setting module 0
Called module parametrised constructor
Called module destructor

... last three lines repeated for each of the modules

Called student destructor
Called module destructor
Called module destructor
Called module destructor
Called module destructor
Called module destructor
Called module destructor
Called programme destructor
```

Listing 8.16: Output from Object Destruction Application

Notice that when we create a new object (the `programme` or `module`) our previously created object goes out of scope and hence the destructor is called. In total, our application will invoke:

- 31 constructor calls
- 31 destructor calls

This is a lot of calls just to create and destroy objects. In practice you should avoid unnecessary object construction and destruction by allocation your objects only once.

8.10.1 Exercise

Again, complete this application, and take note now of when objects are also destroyed. Try and figure out why objects are destroyed when they are. You need to understand that whenever an object is overwritten the previous object is destroyed (unless we are dealing with pointer and references to objects). This is another important realisation when working with data.

8.10.2 Scope Revisited

We have discussed scope a number of times throughout this module and this is another place where scope is very important. Understanding when an object goes out of scope and the impact that has is fundamental for working with a language such as C++. Our memory allocation strategies revolve around scope and therefore you should try and keep track when your objects are created and destroyed.

As soon as an object goes out of scope it is destroyed. There is no delay. There is no garbage collection. Objects are destroyed immediately and the relevant side effects incurred.

8.11 Creating Objects on the Heap

We already covered the general idea of creating objects on the heap using the `new` operator. Let us follow this through with our understanding of object construction and destruction.

Remember that we have two places where our data can be allocated. The stack is where any local values are allocated, and can be considered our working memory. It is cleared up as values go out of scope and does not require any additional work from us as programmers. Up until now we have being constructing objects on the stack.

The heap is our large area of memory where we can allocate data for use outside our local scope, and for values that do not fit in the stack. The heap is where memory is allocated in C++ when we use the `new` operator. When we create an object in this manner we must remember to explicitly call `delete` (unless we are using smart pointers). `new` will call the relevant constructor of the object and `delete` will call the object's destructor.

Let us now change our application so that we allocate the student object on the heap rather than the stack. The following application illustrates this.

```
#include <iostream>
#include <string>

using namespace std;

// Defines a programme of study
struct programme
{
  string code;
  string name;

  // Default constructor
  programme()
  {
    cout << "Called programme default constructor" << endl;
  }

  // Constructor with parameters
  programme(const string &code, const string &name)
  {
    cout << "Called programme parametrised constructor" << endl;
    this->code = code;
    this->name = name;
  }

  // Destructor for programme
  ~programme()
  {
    cout << "Called programme destructor" << endl;
  }
};

// Defines a module
struct module
{
  string code;
  string name;
  unsigned char mark;

  // Default constructor
  module()
  {
    cout << "Called module default constructor" << endl;
  }

  // Constructor with parameters
  module(const string &code, const string &name, unsigned int mark)
  {
    cout << "Called module parametrised constructor" << endl;
    this->code = code;
    this->name = name;
    this->mark = mark;
  }

  // Module destructor
  ~module()
  {
    cout << "Called module destructor" << endl;
  }
};

// Defines a student
struct student
{
  unsigned int matric;
  string name;
  string address;
  programme prog;
  module mods[6];

  // Default constructor
  student()
  {
    cout << "Called student default constructor" << endl;
  }

  // Programme destructor
  ~student()
  {
    cout << "Called student destructor" << endl;
  }
};

int main(int argc, char **argv)
{
  cout << "Creating a student object on the heap" << endl;
  student *s = new student();

  // Set some student values
  cout << "Setting student details" << endl;
  s->matric = 42001290;
  s->name = "Kevin Chalmers";
  s->address = "School of Computing";

  // Set the programme
  cout << "Setting programme" << endl;
  s->prog = programme("56119BH", "BEng (Hons) Software Engineering");

  // Set the modules
  cout << "Setting module 0" << endl;
  s->mods[0] = module("SET07109", "Programming Fundamentals", 85);

  // **** Add other modules ****

```

```
105     // **** Calculate average mark ****
106
107     // **** Print average mark ****
108
109     // Delete the student object
110     cout << "Deleting the student object from the heap" << endl;
111     delete s;
112
113     return 0;
114 }
```

Listing 8.17: Creating Objects on the Heap

Notice that we are using the -> operator again. We will come to this shortly.

Our only difference in this application is that we are using new to create our student object (line 87) and then using delete to explicitly destroy the object on line 111. The output is saved for the exercise.

8.11.1 Exercise

Complete the application again and observe the construction and destruction order. In particular, compare this to the messages generated when we didn't use the stack for creating objects. Reflect on what is happening from the point of view of declaring a pointer to a value and actually calling new and delete.

8.12 Accessing Members of Pointers

We have introduced the -> operator already but now let us explore why we need it. Now that we are creating objects on the heap we have a pointer to the object rather than direct access to the object. This means that the variable we have is not an instance of the object but rather a memory location to where such an object exists. When we introduced pointers we discussed why we needed pointer dereferencing to allow access to the raw object. This is what we need to do now as well.

8.12.1 Dereferencing the Pointer

Remember that when we have a pointer to a value (for example int *x) to access the value we need to use the dereference operator (*) on the variable to gain access to it (for example int y = *x). The same applies if we want to call methods or access attributes of an object.

Let us consider our student class defined previously. Let us say we have a pointer to such a value (let us call it student *s). We can access the individual values of this student by dereferencing the pointer and then getting the value from this dereferenced object.

```
student *s = new student();
// ... do some other work
student a = *s;
programme p = a.prog;
```

This seems like a lot of work to access an individual value (and will also cause a copy of the student object to be created, hence a constructor call and this object to be destroyed hence a destructor call). A better way is to perform these two operations in one line:

```
student *s = new student();
// ... do some other work
programme p = (*a).prog;
```

Notice that we have to surround the dereference operation in brackets to access the members of the object. Let us first use this technique to work with a pointed to object. The code for this example is below.

```
#include <iostream>

using namespace std;

// Simple test class
class my_class
{
public:
  unsigned int x = 10;
  unsigned int y = 20;

  void set_x(unsigned int value)
  {
    x = value;
  }

  void set_y(unsigned int value)
  {
    y = value;
  }
};

int main(int argc, char **argv)
{
  // Allocate an object on the heap
  my_class *a = new my_class();

  // Print values of x and y.
  // Need to dereference the pointer
  cout << "a.x = " << (*a).x << endl;
  cout << "a.y = " << (*a).y << endl;

  // Set the values of x and y
  // Need to dereference the pointer
  (*a).set_x(20);
  (*a).set_y(40);

  // Print values of x and y.
  // Need to dereference the pointer
  cout << "a.x = " << (*a).x << endl;
  cout << "a.y = " << (*a).y << endl;

  delete a;

  return 0;
}
```

Listing 8.18: Accessing Members by Dereferencing a Pointer

You should be able to predict the output from this application by now.

8.12.2 Using the -> Operator

The method of dereference and access takes an additional four characters - and programmers are lazy! Actually a more reasonable reason for not liking this syntax is it is not easy to spot the intention of the programmer and could be missed when looking for bugs. Therefore the dereference and access can be abbreviated into the -> operator. Let us rewrite the application using this different method.

```
 1 #include <iostream>
 2 
 3 using namespace std;
 4 
 5 // Simple test class
 6 class my_class
 7 {
 8 public:
 9   unsigned int x = 10;
10   unsigned int y = 20;
11 
12   void set_x(unsigned int value)
13   {
14     x = value;
15   }
16 
17   void set_y(unsigned int value)
18   {
19     y = value;
20   }
21 };
22 
23 int main(int argc, char **argv)
24 {
25   // Allocate an object on the heap
26   my_class *a = new my_class();
27 
28   // Print values of x and y.
29   // Use nicer syntax
30   cout << "a.x = " << a->x << endl;
31   cout << "a.y = " << a->y << endl;
32 
33   // Set the values of x and y
34   // Need to dereference the pointer
35   a->set_x(20);
36   a->set_y(40);
37 
38   // Print values of x and y.
39   // Need to dereference the pointer
40   cout << "a.x = " << a->x << endl;
41   cout << "a.y = " << a->y << endl;
42 
43   delete a;
44 
45   return 0;
46 }
```

Listing 8.19: Using the -> Operator

All we have done is changed the dereference lines (lines 30,31, 35, 36, 40 and 41) to use our -> operator. This is how you should work when using pointers to objects. This operator is also overloaded for smart pointers so that the -> operator also works there to.

8.13 Inheritance

The final main idea of object orientation we shall cover is the idea of inheritance. Inheritance allows us to `extend` an object's data and behaviour in a subclass. It allows us to treat an objects as classifications.

For example let us consider that we need an application that has objects that are animals. We therefore define a class animal. However, animals can be further classified into say mammals and birds. We can go even further and say that a mammal can be a cat or a dog. With this distinction, we can happily say that a dog is a mammal which is an animal, therefore a dog is an animal.

It may be the case that we have an application that works on a collection of animal objects. It doesn't need to know about the different types of animals, only that animals exist. The animal may be a dog or it may be seagull, or even a human. This doesn't matter to the application - it just needs to know about animals and hence we can treat every object as type animal.

We can also extend this idea further. Let us say we know some properties of that all animals must have such as a name. Therefore we declare that the animal class has a property name which means that mammals and therefore dogs also have names.

We can do the same with methods, and we can also override these methods (this is the next unit on virtual functions). For example, let us state that all animals will have a method `speak` which prints the noise that the animal makes. Obviously we cannot implement this in animal - there is no one sound all animals make. However, we can implement the dog's `speak` method to output "bark". Our application doesn't care - it just knows that animals can speak and can call the method appropriately. The method called is the actual `speak` of that particular type of animal.

Let us do another class diagram for an application this time adding inheritance. Inheritance is sometimes termed *generalization* or *specialization* (depending on the direction you are referring to). We signify that a class is a specialization of another class (the generalization) by drawing an arrow from this specialization to the generalization. Figure 8.2 illustrates this.

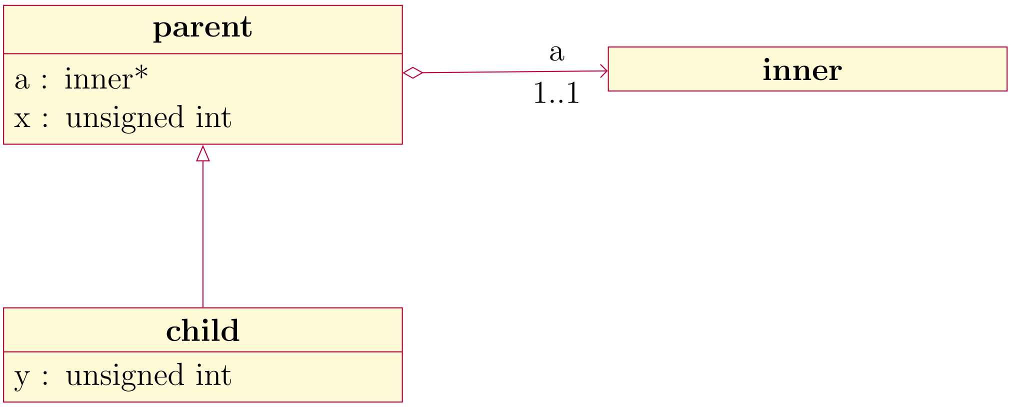


Figure 8.2: Class Diagram Showing Inheritance

Here we are saying that the `child` class is a specialization of the `parent` class. Therefore the `child` class also contains a pointer to the `inner` class and an `unsigned` int called x. The child class also declares another unsigned int called y.

In memory an object with inheritance has its inherited values added first. Therefore in memory our child object is as follows:

Bytes	Data
0 to 3	a
4 to 7	x
8 to 11	y

The more classes inherited from, the more data in the object. Inheriting from another classes requires us to define sub-classes which are done in the class declaration.

Declaring a Sub-class

To declare a class as a sub-class of another we do so after we declare the class name in C++. We do this using a colon and then listing the classes that the object inherits. For example, returning to our animal case study above we would define the following:

```cpp
class animal
{
    // .. animal declaration
};

class mammal : public animal
{
    // ... mammal declaration
};

class dog : public mammal
{
    // ... dog declaration
};
```

Notice the use of the keyword public. This is an important requirement when working with correct object-orientation and inheritance. If the keyword public isn't used then the class contains the members of its parents, but is not of that type (it cannot be cast as such).

Let us now look at an example application using inheritance built from our class diagram. Again, you should try and implement this yourself first from the diagram if you can and then check your understanding from the code provided.

```cpp
#include <iostream>

using namespace std;

// Inner class
class inner
{
public:
  inner()
  {
    cout << "Called inner constructor" << endl;
  }
  ~inner()
  {
    cout << "Called inner destructor" << endl;
  }
};

class parent
{
private:
  // Parent contains a pointer to an inner object
  inner *a;
public:
  // Parent contains an unsigned int value
  unsigned int x;

  parent()
  {
    cout << "Called parent constructor" << endl;
    // Calls default constructor
    a = new inner;
  }
  ~parent()
  {
    cout << "Called parent destructor" << endl;
    delete a;
  }
};

class child : public parent
{
public:
  // Child adds an unsigned int value
  unsigned int y;

  child()
  {
    cout << "Called child constructor" << endl;
  }
  ~child()
  {
    cout << "Called child destructor" << endl;
  }
};

int main(int argc, char **argv)
{
  // Print out size of inner
  cout << "sizeof(inner) = " << sizeof(inner) << " bytes" << endl;
  // Print out size of parent
  cout << "sizeof(parent) = " << sizeof(parent) << " bytes" << endl;
  // Print out size of child
  cout << "sizeof(child) = " << sizeof(child) << " bytes" << endl;

  // Create a child object on the heap
  // Calls default constructor
  child *c = new child;

  // Delete the child object
  delete c;

```

```
73   return 0;
74 }
```

Listing 8.20: Inheritance in C++

The following lines are of interest:

line 6 - class `inner` has no attributes. Although you may think it is 0 bytes in memory, C++ will make it 1 byte just so it has a memory location.

line 23 - here we are declaring that we have a pointer to a `inner` object. Therefore this only take up 4 (or 8 in 64-bit applications) bytes.

line 32 - we need to allocate memory for our `inner` object. We do this using `new`. This is because we have a pointer.

line 37 - we need to remember to call `delete` on our `inner` object in the destructor to free up memory.

line 41 - here we have declared `child` as a specialization of `parent`.

Type Sizes with Inheritance

A class that inherits from another class also contains its attributes. As such, any further attributes defined for the sub-class will add to this size.

Although it is unlikely you will every create a type which has a problem with its size, it is worth keeping track of your objects are stored in memory. This allows you to undertake some low level debugging in memory if required.

Multiple Inheritance

It is possible to have a class that inherits from multiple super-classes. This is something we term multiple inheritance. This allows us to define a type as implementing the members of multiple classes.

In C++ multiple inheritance acts the same - we just list more `public` classes after the name. This form of inheritance is not possible in Java and C#. These languages only allow one class to be inherited from. They get around the use of multiple inheritance by using *interfaces*. Interfaces provide much of the same concepts as standard inheritance except that an interface cannot define data members (attributes) or implemented methods. C++ has no concept of interfaces, although pure virtual classes do provide some of the functionality.

Construction and Destruction Order with Inheritance

We are still concerned with object construction when working with inheritance. Any specialization that we have must also initialise its super-class attributes. When an object is constructed these are done first in order of their declaration. We have the following order of operations:

1. The memory required for the object being created is allocated
2. The parent classes of the object being created are initialised in the order they are declared. The programmer can dictate which constructor should be called if the default one is not required

3. The body of the constructor for the object being created is invoked

As before, destruction happens in the opposite order:

1. The body of the destructor for the object being destroyed is invoked
2. The parent classes of the object being destroyed are called in the opposite order they are declared
3. The memory required for the object is freed

The complete object construction order is therefore the following:

1. The memory required for the object being created is allocated
2. The parent classed of the object being created are called in the order they are declared
3. The attributes of the object are created in the order that they are declared
4. The body of the constructor is invoked

Destruction order is this in reverse.

The output for this application is as follows:

```
sizeof(inner) = 1 bytes
sizeof(parent) = 8 bytes
sizeof(child) = 12 bytes
Called parent constructor
Called inner constructor
Called child constructor
Called child destructor
Called parent destructor
Called inner destructor
```

Listing 8.21: Output from Inheritance Application

Notice that `inner` is 1 byte in size (as discussed above), otherwise the size of `child` is the size of `parent` plus the 4 bytes required for the `unsigned int`. Also note the construction and destruction order.

8.13.1 Exercise

Modify the application so that you declare the value `c` as a pointer to a `parent`. Still create a `child` object, just declare it as a `parent`. Now observe what happens when you call `delete`. Then change the declaration of the `parent` destructor to the following:

```
virtual  parent() // body as before
```

What is the output now? What does the `virtual` keyword appear to do? Don't worry if you can't quite get it - we will cover this in full in the next unit.

8.14 protected Class Members

Now that we have introduced inheritance to our application we can describe the other accessibility modifier provided in C++ - protected. As with public and private, protected allows us to determine how a class member is visible. This is neither public (cannot be accessed outside the object), but unlike private does allow sub-classes access to the variable.

The protected Accessibility Modifier

protected allows us to state that a member is visible to sub-classes of a class, but not outside a class. They are good for providing attributes and methods which a sub-class needs access to in its own methods. Although we have stated a sub-class contains its parents values, these are not visible to the sub-class unless they are declared public or private.

To help illustrate these three accessibility methods consider the following table:

Modifier	Visible Object	Visible Sub-class	Visible Outside
public	yes	yes	yes
private	yes	no	no
protected	yes	yes	no

public, private, and protected provide us with the three levels of accessibility that we need for object members in our applications.

Let us now build a test application to see what happens we try and access this new level of accessibility in different parts of our application.

```
#include <iostream>

using namespace std;

class parent
{
private:
  // This is only accessible in parent
  unsigned int x;
protected:
  // This is accessible in parent and its children
  unsigned int y;
public:
  // This is accessible in parent, children and externally
  unsigned int z;

  void print_parent()
  {
    cout << "parent = {" << x << "," << y << "," << "," << "}" <<
        endl;
  }
};

class child : public parent
{
private:
  // This is only accessible in child
  unsigned int a;
protected:
  // This is accessible in child and its children
  unsigned int b;
public:
  // This is accessible in child, children and externally
  unsigned int c;

  void print_child()
  {
    // Parent details - x not accessible.  Won't compile
    cout << "parent = {" << x << "," << y << "," << z << "}" << endl;
    // Child details
    cout << "child = {" << a << "," << b << "," << c << "}" << endl;
  }
};

int main(int argc, char **argv)
{
  // Print sizeof parent
  cout << "sizeof(parent) = " << sizeof(parent) << endl;
  // Print sizeof child
  cout << "sizeof(child) = " << sizeof(child) << endl;

  // Create a child
  child ch;

  // Access values of ch
  // Won't compile - not accessible
  cout << "ch.x = " << ch.x << endl;
  // Won't compile - not accessible
  cout << "ch.y = " << ch.y << endl;
  // Will compile - z publicly accessible
  cout << "ch.z = " << ch.z << endl;
  // Won't compile - not accessible
  cout << "ch.a = " << ch.a << endl;
  // Won't compile - not accessible
  cout << "ch.b = " << ch.b << endl;
  // Will compile - c publicly accessible
  cout << "ch.c = " << ch.c << endl;

  // Call print parent on ch
  ch.print_parent();
  // Call print child on ch
  ch.print_child();

  return 0;
}
```

Listing 8.22: `protected` Class Members

The compiler will output the familiar not accessible errors we had before when working with `private` except the error will be about `protected`:

```
C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\INCLUDE\xlocale(337) : wa
rning C4530: C++ exception handler used, but unwind semantics are not enabled. S
pecify /EHsc
protected.cpp(38) : error C2248: 'parent::x' : cannot access private member decl
ared in class 'parent'
        protected.cpp(9) : see declaration of 'parent::x'
        protected.cpp(6) : see declaration of 'parent'
protected.cpp(56) : error C2248: 'parent::x' : cannot access private member decl
ared in class 'parent'
        protected.cpp(9) : see declaration of 'parent::x'
        protected.cpp(6) : see declaration of 'parent'
protected.cpp(58) : error C2248: 'parent::y' : cannot access protected member de
clared in class 'parent'
        protected.cpp(12) : see declaration of 'parent::y'
        protected.cpp(6) : see declaration of 'parent'
protected.cpp(62) : error C2248: 'child::a' : cannot access private member decla
red in class 'child'
        protected.cpp(27) : see declaration of 'child::a'
        protected.cpp(23) : see declaration of 'child'
protected.cpp(64) : error C2248: 'child::b' : cannot access protected member dec
lared in class 'child'
        protected.cpp(30) : see declaration of 'child::b'
        protected.cpp(23) : see declaration of 'child'
```

Listing 8.23: Compiler Output from `protected` Application

8.14.1 Exercise

Fix this application by introducing a method that allows accessing of the values in the class. These methods will need to be `public`.

8.15 For the Brave - `static` `class` Members

We came across `static` in our work with C. Remember that a `static` value in a function was one that was not deleted when our function went out of scope and we could retain the value through success calls to the function (see Listing 2.20).

In object-orientated terms we can also declare `static` values of a class. They also have runtime life within our applications. Note however that a `static` value belongs to the class, not an instance of the class (an object). That is, a `static` value is an attribute of the class definition, not an object.

`static` allows us to have values that we consider shared amongst all instances of an object. They also allow us to have values that exist throughout the application. At present they may not seem that useful, but when writing larger applications, and in particular using design patterns, you will find `static` a useful feature.

`static` can also be used with methods, allowing us to create a method that belongs to the class rather than an object. This gives us a method that can be called without creating an instance of the class. In fact, this is how Java and C# enable main methods within objects by declaring a `static` main method. In Java, you should have written:

```
public static void main(String[] args)
```

You can see that this method is:

- publicly accessible outside the class

- is `static` so can be called without creating an instance of the class. As this is a main function no objects will have been created

- returns `void` (nothing). This is different than our C++ applications

- takes an array of `String` as arguments - these are the command line arguments

C++ is not just a object-oriented language, and therefore we can define functions outside of classes. Therefore our main methods are not `static`.

`static` in `class` Definitions

To declare a member of a class as `static` we simply use the keyword before the declaration of the member. For example:

```
class my_class
{
private:
    // x is a static value - belongs to the class
    static int x;
public:
    // get_x is a static method - belongs to the class
    static int get_x() { return x; }
};
```

To access a `static` member we use the name of the class followed by two colons and the name of the member. For example to call `get_x` above we use the following:

```
int y = my_class::get_x();
```

To illustrate the use of `static` in a class, and how we access a `static` member, consider the following application:

```
#include <iostream>

using namespace std;

class my_class
{
private:
  // This value is stored in the class - not an object
  static unsigned int x;
public:
  // This method is called via the class
  static unsigned int get_x()
  {
    return x;
  }
  // This method is called via the class
  static void set_x(unsigned int value)
  {
    x = value;
  }
};

// We have to initialise my_class::x
unsigned int my_class::x = 10;

int main(int argc, char **argv)
{
  // Print the sizeof my_class - x is not counted
  cout << "sizeof(my_class) = " << sizeof(my_class) << endl;

  // Print x
  cout << "my_class::x = " << my_class::get_x() << endl;
  // Set x
  my_class::set_x(20);
  // Print x
  cout << "my_class::x = " << my_class::get_x() << endl;

  return 0;
}
```

Listing 8.24: Defining and Using `static class` Members

The output from this application is as follows:

```
sizeof(my_class) = 1
my_class::x = 10
my_class::x = 20
```

Listing 8.25: Output from `static` Application

8.16 For the Brave - Casting with `dynamic_cast` and `reinterpret_cast`

C++ provides two other types of casting when working with objects - `dynamic_cast` and `reinterpret_cast`. These two types of casting are used to cast an object from one type to another in different ways.

`dynamic_cast` is used to *safely* cast from one type to a parent or child type. It does this by checking that the type is correct, and if not returning null (0) as the memory address.

`reinterpret_cast` is the opposite and just treats the memory as the type given. This is fast but unsafe - we are effectively going back to C style casting here.

Both of these cast types take pointers as their parameters, and also return pointers. Otherwise they are similar to `static_cast` and `const_cast`. The following application illustrates:

```
#include <iostream>

using namespace std;

// Inner class
class inner
{
public:
  inner()
  {
    cout << "Called inner constructor" << endl;
  }
  virtual ~inner()
  {
    cout << "Called inner destructor" << endl;
  }
};

class parent
{
private:
  // Parent contains a pointer to an inner object
  inner *a;
public:
```

```cpp
  // Parent contains an unsigned int value
  unsigned int x;

  parent()
  {
    cout << "Called parent constructor" << endl;
    // Calls default constructor
    a = new inner;
  }
  virtual ~parent()
  {
    cout << "Called parent destructor" << endl;
    delete a;
  }
};

class child : public parent
{
public:
  // Child adds an unsigned int value
  unsigned int y;

  child()
  {
    cout << "Called child constructor" << endl;
  }
  ~child()
  {
    cout << "Called child destructor" << endl;
  }
};

int main(int argc, char **argv)
{
  // Create a child class.  Use parent as type
    parent *p = new child();
    // Create a new inner class
    inner *i = new inner();

    // Convert p to a child using dynamic_cast
    child *c = dynamic_cast<child*>(p);
    // Print out memory location
    cout << "Memory location of p dynamically cast to child: " << c << endl;
    // Do the same for inner
    c = dynamic_cast<child*>(i);
    // Print out memory location
    cout << "Memory location of i dynamically cast to child: " << c << endl;

    // Now do the same with reinterpret_cast
    c = reinterpret_cast<child*>(p);
    cout << "Memory location of p reinterpreted to child: " << c << endl;
    c = reinterpret_cast<child*>(i);
    cout << "Memory location of i reinterpreted to child: " << c << endl;

  // Delete objects
  delete i;
```

```
    delete p;

  return 0;
}
```

Listing 8.26: `dynamic_cast` and `reinterpret_cast` Test Application

Notice that we declare the destructors of `parent` and `inner` as `virtual`. This is to signify that the types are *polymorphic* (can be cast). An example output from this application is as follows:

```
Called parent constructor
Called inner constructor
Called child constructor
Called inner constructor
Memory location of p dynamically cast to child: 01024A50
Memory location of i dynamically cast to child: 00000000
Memory location of p reinterpreted to child: 01024A50
Memory location of i reinterpreted to child: 0102BFA0
Called inner destructor
Called child destructor
Called parent destructor
Called inner destructor
```

Listing 8.27: Output from Casting Application

Notice that our `dynamic_cast` from `inner` to `child` provided us with the memory location 0 (or null). Therefore, C++ could not cast `inner` to `child`. This is something we can check in our code to determine if a value is of a particular type.

Also notice that our `reinterpret_cast` caused a different memory location for `inner` when casting to `child`. What has happened is that a copy of the data was created to allow it to be treated as an `inner`.

Casting between object types is a very common requirement in object-orientation. C++ does not naturally retain type information, and therefore doing these types of casts requires *Run-Time Type Information* (RTTI). This has an impact on performance and is therefore frowned upon in certain areas of software development such as games.

8.17 For the Brave - Writing and Reading `struct` and `class` Data

Storing data in our code in objects is useful, but what about storing these objects on file? Well, we can do this in C++ if we treat the objects as raw data. This is the simplest method, but modern approaches use techniques called *object serialization* or we could store them in a database. These ideas are well outside the scope of this module as we will just look at raw output.

The following code illustrates the technique. Here we just treat the memory as character data and write and read accordingly.

```
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

// The struct we will write out
struct employee
{
    unsigned int number;
    char name[100];
    char job_title[100];
    float salary;

    void print()
    {
        cout << number << " " << name << " " << job_title << " " << salary << endl;
    }
};

void write_employee(employee *emp, const string &filename)
{
    // Open file for writing
    ofstream output;
    output.open(filename.c_str(), ios::out | ios::app | ios::binary);
    // Write employee - note the cast to char*
    output.write((char*)(emp), sizeof(employee));
    output.close();
}

void read_employees(vector<employee> &emp, const string &filename)
{
    // Open file for reading
    ifstream input;
    input.open(filename.c_str(), ios::in | ios::binary);
    // Loop until end of file
    employee e;
    while(input.read((char*)&e, sizeof(employee)))
        emp.push_back(e);
}

int main(int argc, char **argv)
{
    // Loop until number is 0
    while (true)
    {
        employee e;
        cout << "Employee number: ";
        cin >> e.number;
        if (e.number == 0)
        break;
        cout << "Employee name: ";
        cin >> e.name;
        cout << "Employee job: ";
        cin >> e.job_title;
        cout << "Employee wage: ";
        cin >> e.salary;
        write_employee(&e, "employees.dat");
    }

    vector<employee> emp;
    read_employees(emp, "employees.dat");

    for (auto &e : emp)
        e.print();

    return 0;
}
```

Listing 8.28: Reading and Writing `struct` Data

The limitation of this approach is that we cannot handle pointers and references. The raw data approach takes what is in memory and dumps it to a file. As a pointer is just a memory location, this is what we dump. That memory location will likely be no longer valid when we read back in the file.

8.18 Exercises

1. Build an application that has a `Pegasus` class. `Pegasus` is a special type of animal. It inherits from both horse and bird. For the complete application you should have the following classes:

   - Animal
   - Mammal
   - Bird
   - Horse
   - Pegasus

   The Animal class should define the following methods:

   - Move
   - Speak
   - Eat

   This is trickier than it sounds because of multiple-inheritance.

2. Create an application with a `car` class. A `car` has multiple parts - wheels (four of), engine, doors, steering wheel, etc. Define the correct class and a necessary test application.

Unit 9

Virtual Function Calls

So far we have covered the following areas in the module:

1. How we compiling and linking programs

2. How our data is represented in memory

3. How our code converts to machine instructions

4. How the pre-processor creates our compilation units

5. How functions are called and parameters passed to them

6. How memory works in our applications

7. How we can debug our applications

8. How we can build more complex applications and data types using object-orientation

In this unit we are going to extend our understanding of object-orientation by looking at virtual functions. Virtual functions are ones that we state can be overridden by our sub-classes, thus providing different behaviour for our sub-classes. This becomes a useful abstraction when we want to treat a collection of objects as the same type, but expect their behaviour to be different.

9.1 Overridable Behaviour in Classes

We already touched on the idea of over-ridable behaviour in one of the previous exercises when we cast our objects between different types. Let us now look at a more concrete example where we can see a problem arising.

In this application we are going to define two classes - a `parent` and a `child`. We will call a method `print` on these objects. What we are going to do is cast a `child` object to a `parent` object and call this method. Our intention is that the `print` method of `child` is called (it is a child object still, we are just thinking of it as a parent - remember the animal-mammal-dog example). Below is this application.

```
#include <iostream>

using namespace std;

class parent
```

```cpp
{
public:
  void print()
  {
    cout << "Calling parent.print" << endl;
  }
};

class child : public parent
{
public:
  void print()
  {
    cout << "Calling child.print" << endl;
  }
};

int main(int argc, char **argv)
{
    // Create a parent object
    parent p;
    // Call print
    cout << "Calling print on a parent" << endl;
    p.print();

    // Create a child object
    child c;
    // Call print
    cout << "Calling print on a child" << endl;
    c.print();

    // Create a child on the heap
    child *c2 = new child();
    // Call print on c2
    cout << "Calling print on a child" << endl;
    c2->print();

    // Cast child to a parent
    // Use dynamic_pointer_cast for smart pointers
    parent *p2 = (parent*)c2;
    // Call print on p2
    cout << "Calling print on a child cast to a parent" << endl;
    p2->print();

    delete c2;

    return 0;
}
```

Listing 9.1: Trying to Override Class Behaviour

When you run this application you will get the following output.

```
Calling print on a parent
Calling parent.print
Calling print on a child
Calling child.print
Calling print on a child
Calling child.print
Calling print on a child cast to a parent
Calling parent.print
```

Listing 9.2: Output from First Override Method Application

So we have a problem here. Notice the last line of output. Here, we have cast the `child` object to a `parent` (line 45), but when we call `print` the `parent` version is called, not the `child` version. This is not what we want. We wanted the object to still behave like a `child`, we just wanted to consider it a `parent`.

9.2 The `virtual` Keyword in C++

So how do we ensure the output from our application is correct? What we really wanted to happen above was the following:

```
Calling print on a parent
Calling parent.print
Calling print on a child
Calling child.print
Calling print on a child
Calling child.print
Calling print on a child cast to a parent
Calling child.print
```

Listing 9.3: Expected Output from First Override Method Application

To do this in C++ we have to use the `virtual` keyword. `virtual` marks a method as overridable. If a method is not marked with `virtual` then it doesn't matter that we override it in a child class, *the compiler doesn't know that this is possible*. Therefore we have to use `virtual` to indicate to the compiler that this is the case.

There is a good chance you have come across similar keywords in Java. Java uses the `abstract` keyword to indicate a method as overridable, although it is not quite the same. In Java, any method can be overridden unless it has been declared `final`. An `abstract` method is one that has no definition. We will discuss the equivalent in C++ (*pure virtual methods*) later in this unit.

9.3 `virtual` Methods

Let us now rewrite our test application using the `virtual` keyword. This is below. Note that the `parent` class has `print` declared as `virtual`. The `child` does not need this as it inherits the property from the `parent`.

```cpp
#include <iostream>

using namespace std;

class parent
{
public:
  virtual void print()
  {
    cout << "Calling parent.print" << endl;
  }
};

class child : public parent
{
public:
  void print()
  {
    cout << "Calling child.print" << endl;
  }
};

int main(int argc, char **argv)
{
  // Create a parent object
  parent p;
  // Call print
  p.print();

  // Create a child object
  child c;
  // Call print
  c.print();

  // Create a child on the heap
  child *c2 = new child();
  // Call print on c2
  c2->print();

  // Cast child to a parent
  parent *p2 = (parent*)c2;
  // Call print on p2
  p2->print();

  delete c2;

  return 0;
}
```

Listing 9.4: Using `virtual` to Declare Overridable Behaviour

The output from this application is now as expected:

```
Calling parent.print
Calling child.print
Calling child.print
Calling child.print
```

Listing 9.5: Output from Correct `virtual` Application

9.4 `virtual` Method Tables

When we declare a function as `virtual` the C++ compiler has to do some extra work. This is to ensure that the correct version of the function is called from an object. This is achieved through the use of *virtual method tables*. A virtual method table is just a collection of pointers that point to the correct version of the methods that should be called (the methods are stored in memory after all). Each type that has `virtual` methods has a virtual method table associated with it. An instance of that type (an object) contains a pointer to the relevant virtual method table.

As an example consider Figure 9.1. This diagram shows that we have three objects (on the left hand side of the diagram). These objects are all inherited from type animal. As such, they have a pointer to a table of functions (the `v_ptr` value). These tables are dependant on the type of the object. The top object is a `Dog` and therefore has a pointer to the `Dog` virtual method table. The other two objects are of type `Duck` and therefore have a pointer to the `Duck` virtual method table.

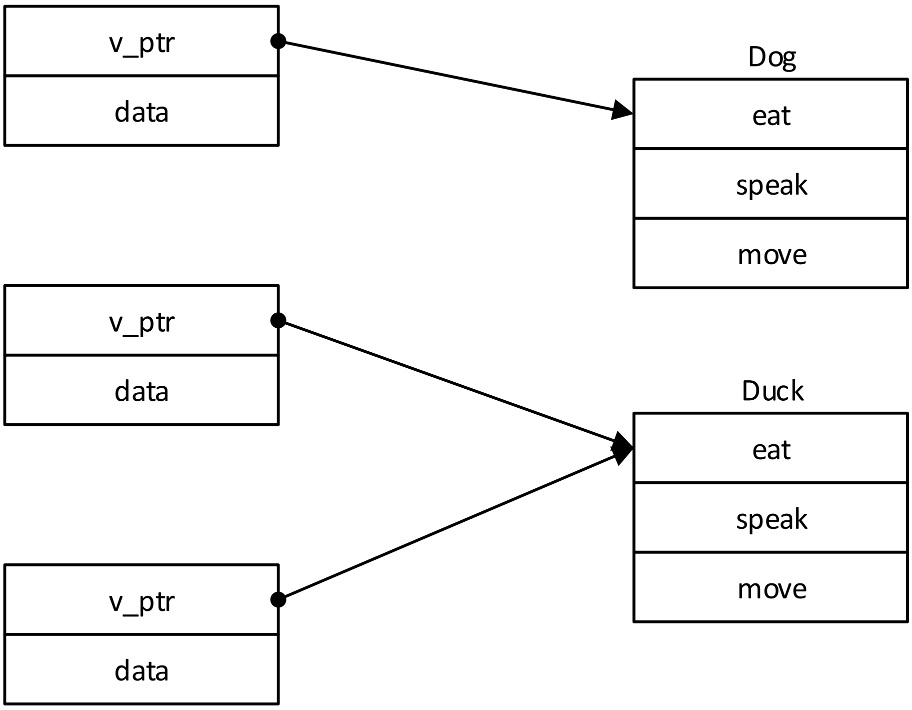


Figure 9.1: Virtual Method Table

Therefore in C++ an object might be an additional 4 bytes (or 8 bytes in 64-bit) in size. These additional bytes *are not* counted in the `sizeof` function. These are additional to the standard data used to represent the class.

9.5 Inheritance and Destruction

Now we come to a very important part of working with inheritance in C++ - *destruction*. As we have seen, if we want to call the correct method on a derived (inherited) object we have to mark the method as `virtual`. So what about for objects allocated on the heap? How do we ensure these are deleted correctly?

In C++, inheritance destruction occurs as follows:

1. The destructor of the object is called
2. The destructor of the parent objects are called

Let us test this idea with an updated application. This is shown below:

```cpp
#include <iostream>

using namespace std;

class parent
{
public:
  // Constructor
  parent()
  {
    cout << "Called parent constructor" << endl;
  }
  // Destructor
  ~parent()
  {
    cout << "Called parent destructor" << endl;
  }
};

class child : public parent
{
public:
  // Constructor
  child()
  {
    cout << "Called child constructor" << endl;
  }
  // Destructor
  ~child()
  {
    cout << "Called child destructor" << endl;
  }
};

int main(int argc, char **argv)
{
  // Create a child on the heap
  child *a = new child;
  // Destroy the child
  delete a;

  // Create a child on the heap
  // Treat as parent
  parent *b = new child;
  // Destroy the child
  delete b;

  return 0;
}
```

Listing 9.6: Destructors with Inheritence

The output from this application is below:

```
Called parent constructor
Called child constructor
Called child destructor
Called parent destructor
Called parent constructor
Called child constructor
Called parent destructor
```

Listing 9.7: Output from non-`virtual` Destructor Application

OK, we have a problem - the last `child` destructor was not called. Only the destructor for `parent` was called. This could cause problems such as memory leaks. We can solve this by declaring the destructor `virtual`.

Destruction Order Overview

We discussed this in the last unit, but just to reiterate here. Destruction order takes place as follows.

1. The body of the destructor is invoked
2. The attributes of the object are destroyed in the reverse order that they are declared

3. The parent classes of the object being destroyed are called in the reverse order they are declared

4. The memory required for the object being created is freed

Without a virtual destructor step 3 is missed causing problems.

9.6 virtual Destructors

Our test application rewritten with a virtual destructor is as follows.

```
#include <iostream>

using namespace std;

class parent
{
public:
  // Constructor
  parent()
  {
    cout << "Called parent constructor" << endl;
  }
  // Destructor
  virtual ~parent()
  {
    cout << "Called parent destructor" << endl;
  }
};

class child : public parent
{
public:
  // Constructor
  child()
  {
    cout << "Called child constructor" << endl;
  }
  // Destructor
  ~child()
  {
    cout << "Called child destructor" << endl;
  }
};

int main(int argc, char **argv)
{
  // Create a child on the heap
  child *a = new child;
  // Destroy the child
  delete a;

  // Create a child on the heap
  // Treat as parent
  parent *b = new child;
  // Destroy the child
  delete b;
```

```

    return 0;
}
```

Listing 9.8: `virtual` Destructors

The output this time is correct, calling all the relevant destructors.

```
Called parent constructor
Called child constructor
Called child destructor
Called parent destructor
Called parent constructor
Called child constructor
Called child destructor
Called parent destructor
```

Listing 9.9: Correct Output with `virtual` Destructors

9.7 Overriding Methods

So we can now override behaviour in sub-classes. This is an important concept in the abstraction that object-orientation provides. It enables us to treat objects as particular high-level types while implementing derived behaviours. This will become a key factor as you develop your object-oriented skills in the future.

However, allowing developers to just override behaviour at will is not always a good idea. Therefore, we will look at how we can control the behaviour of overriding, and allow the compiler to help us if we have made a mistake. We will introduce two new keywords for this - `override` and `final`.

9.7.1 `override` Keyword

The `override` keyword allows us to declare a method as overriding a `virtual` method. So what you may ask - we can do this already. Well the `override` keyword allows us to specify that this was our intention. Therefore, if we attempt to override a method that we cannot, the compiler will output an error. This is how you should declare overriding methods as standard to improve overall code quality.

Declaring `override` Methods

The `override` keyword used as a specifier on a method. As such, it follows the definition of the method in the class declaration. As mentioned, the `override` keyword is really just an aid to the compiler to make sure you have declared your inheritance correctly.

Declaring `override` Methods

To declare a method as override consider the following example:

```
class A
{
    virtual void foo();
    void bar();
};

class B : public A
```

```
{
    void foo();
    void bar();
};
```

The above example will compile quite happily in standard C++. The foo method is virtual, and therefore the subclass B is providing a different implementation of this function. For the bar method the one declared in B is not overriding behaviour as such. Although when you have an object of type B you can call bar, you cannot cast the object to type A and get the same behaviour - the method is not virtual.

Now consider the addition of the override keyword.

```
class A
{
    virtual void foo();
    void bar();
};

class B : public A
{
    void foo() override;
    void bar() override;
};
```

This time the compiler will throw an error - the bar method is not virtual and therefore cannot be overridden. This is the key point of the override keyword (as well as potentially allowing the compiler to optimise your code).

To test the override keyword let us look at the following example.

```
#include <iostream>

using namespace std;

class parent
{
public:
  virtual void my_method()
  {
    cout << "Called parent.my_method()" << endl;
  }
};

class child : public parent
{
public:
  void my_method() override
  {
    cout << "Called child.my_method()" << endl;
  }
};

int main(int argc, char **argv)
{
  // Create a parent object on the heap
  parent *p = new parent;
  // Call my_method
28   p->my_method();
29
30   // Create a child object on the heap
31   child *c = new child;
32   // Call my_method
33   c->my_method();
34
35   // Delete parent
36   delete p;
37
38   // Set p to c
39   p = (parent*)c;
40
41   // Call my_method
42   p->my_method();
43
44   delete c;
45
46   return 0;
47 }
```

Listing 9.10: Using the `override` Keyword

On line 17 we have declared our `my_method` function with `override`. This means that we are telling the constructor that we believe this is an overriding function from the parent class. In this instance we are correct, and running the application will give the following:

```
Called parent.my_method()
Called child.my_method()
Called child.my_method()
```

Listing 9.11: Output from First `override` Application

Exercise Modify the previous application to take away the `virtual` keyword and try and compile again. What is the compiler output?

Multilevel Inheritance

With multi-level inheritance, the `override` specifier allows us to denote that we are intending to override a particular base behaviour from somewhere further up the inheritance hierarchy. We use this to keep track of what we are doing in code.

As an example, consider Figure 9.2. Here we have an application with three levels of inheritance.

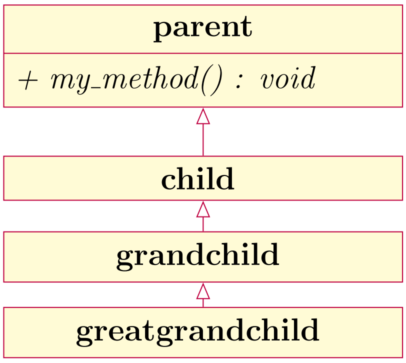


Figure 9.2: Great Grandchild Application Diagram

The code for this application is below. Notice that `grandchild` does not implement anything, yet we specify that `great_grandchild` does `override`. This is fine in our hierarchy.

```cpp
#include <iostream>

using namespace std;

class parent
{
public:
  virtual void my_method()
  {
    cout << "Called parent.my_method()" << endl;
  }
};

class child : public parent
{
public:
  void my_method() override
  {
    cout << "Called child.my_method()" << endl;
  }
};

class grandchild : public child
{
  // Not declaring anything
};

class great_grandchild : public grandchild
{
public:
  void my_method() override
  {
    cout << "Called great_grandchild.my_method()" << endl;
  }
};

int main(int argc, char **argv)
{
  // Create all object types on the heap
  parent *p[4];
  p[0] = new parent;
  p[1] = new child;
  p[2] = new grandchild;
  p[3] = new great_grandchild;

  // Call my_method on each
  for (int i = 0; i < 4; ++i)
    p[i]->my_method();

  // delete all the objects
  for (int i = 0; i < 4; ++i)
    delete p[i];

  return 0;
}
```

Listing 9.12: `override` and Multilevel Inheritance

The output from this application is as follows:

```
Called parent.my_method()
Called child.my_method()
Called child.my_method()
Called great_grandchild.my_method()
```

Listing 9.13: Output from Multiple Inheritance Application

Notice that the call for `grandchild->my_method` just use the `child` version of the method behaviour. The overridden `great_grandchild` uses its own behaviour.

Why we Need `override`

As mentioned, `override` is used to help the compiler spot errors that we may have in our code to do with how we want our application to behave. These bugs are very subtle and can leave a programmer frustrated for a number of hours trying to spot why a particular function is not behaving correctly. As such, we should use `override` as much as possible.

The following code sample helps illustrate what happens when we inadvertently hide behaviour of our application.

```
#include <iostream>

using namespace std;

class base
{
public:
    virtual void print(unsigned int x)
    {
      cout << "Calling base.print with " << x << endl;
    }
};

class derived1 : public base
{
public:
    // Compiling this will cause an error - no override
    void print(float x) override
    {
        cout << "Calling derived1.print with " << x << endl;
    }
};

class derived2 : public base
{
public:
    // Compiling this will not cause an error - hides previous behaviour
    void print(float x)
    {
        cout << "Calling derived2.print with " << x << endl;
    }
};

int main(int argc, char **argv)
{
    base *b1 = new derived1;
    base *b2 = new derived2;

39     b1->print(20.0f);
40     b2->print(30.0f);
41 
42     delete b1;
43     delete b2;
44 
45     return 0;
46 }
```

Listing 9.14: Hiding Behaviour

Notice on line 18 that we override the behaviour defined in `base` but using a different parameter type (`derived1` expects a float, not an `int`). The compiler error is as follows:

```
override3.cpp(18) : error C3668: 'derived1::print' : method with override specif
ier 'override' did not override any base class methods
```

Listing 9.15: Compiler Error for Incorrect `override`

Again we are letting the compiler do the work for us. This is the biggest advantage of a compiled language - better static analysis of your code to ensure it is correct before you even run it. The more bugs caught at compile time the better.

9.7.2 `final` Keyword

C++ has another specifier for methods in regards to inheritance and `virtual` definition - `final`. The keyword `final` is used to denote that a method *should not be* overridden. This is useful in API design when you want to ensure that behaviour is not overridden by sub-classes. This becomes more apparent as you start to develop larger applications.

The following application illustrates what happens when using `final` and trying to violate that statement.

```
#include <iostream>

using namespace std;

class parent
{
public:
  virtual void my_method()
  {
    cout << "Called parent.my_method()" << endl;
  }
};

class child : public parent
{
public:
  void my_method() override final
  {
    cout << "Called child.my_method()" << endl;
  }
};

class grandchild : public child
{
public:
  // This won't compile.  Cannot override a final
  void my_method() override
  {
    cout << "Called grandchild.my_method()" << endl;
  }
};

int main(int argc, char **argv)
{
  // Create a grandchild object
  parent *p = new grandchild;

  // Call my_method
  p->my_method();

  // Delete object
  delete p;
}
```

Listing 9.16: Using the `final` Keyword

`final` on any Method

As your skills in object-orientated development improve you should start considering more the use of `final`. A good default approach is to define any method as `final` as default and remove it only when you are sure that you don't need it anymore and need to override the default behaviour. Again, it is just a technique to improve your development and avoid bugs (such as running functions that have overridden expected behaviour).

Trying to compile this application will give the following output:

```
final.cpp(27) : error C3248: 'child::my_method': function declared as 'final' ca
nnot be overridden by 'grandchild::my_method'
        final.cpp(17) : see declaration of 'child::my_method'
```

Listing 9.17: Compiler Output from Trying to `override` `final` Methods

9.8 Pure `virtual` Methods

So far our use of `virtual` has been about overriding default behaviour in a base class. But what if there is no default behaviour? What do we do then? This is where we need what is known as *pure virtual methods*.

A pure virtual method is one where we define no behaviour - the declaration of the method in the base class has no body. This means that the base class is something we define as *abstract*. An abstract class is one which we cannot create instances of (they are missing some part of their implementation - the pure virtual method). We can only create a derived class from this base class if it has the necessary pure virtual method implemented.

Think about our previous animal based example. We have defined a base class - `animal` - which has a method `speak`. In our previous examples, we had to implement some default behaviour for `speak` - we printed `animal noise`. Obviously this is not accurate - there is no default noise all animals make.

If we introduce pure virtual methods we can get round the problem by stating that an `animal` has a method `speak` but no more. This allows us to define certain behaviours that all animals should have without having to think of a default behaviour.

Declaring a Method as Pure Virtual

Declaring a method as pure virtual is easy - we just say that the method is equal to 0:

```
virtual void my_method() = 0;
```

The compiler takes the 0 to mean that no definition is made for this class and the class should also be considered abstract.

Let us look at a test application. Below we define a pure virtual method in the `parent` class, and then override it in the `child` class.

```
 1 #include <iostream>
 2 
 3 using namespace std;
 4 
 5 class parent
 6 {
 7 public:
 8     virtual void my_method() = 0;
 9 };
10 
11 class child : public parent
12 {
13 public:
14     void my_method()
15     {
16         cout << "Called child.my_method()" << endl;
17     }
18 };
19 
20 int main(int argc, char **argv)
21 {
22     // Try and create a parent - won't compile
23     parent *p = new parent;
24     // Create a child - no problems
25     parent *c = new child;
26 
27     // Call my_method
28     c->my_method();
29 
30     // Delete the objects
31     delete p;
32     delete c;
33 
34     return 0;
35 }
```

Listing 9.18: Pure `virtual` Functions

Notice that we try and create an instance of `parent` on line 23. This isn't possible as the compiler knows the class is abstract. Trying to compile this application will output the following compiler error:

```
pure_virtual.cpp(23) : error C2259: 'parent' : cannot instantiate abstract class
```

```
        due to following members:
        'void parent::my_method(void)' : is abstract
        pure_virtual.cpp(8) : see declaration of 'parent::my_method'
```

Listing 9.19: Compiler Error Trying to Create Abstract Class

9.8.1 Exercise

Fix the above application so that it will compile.

9.9 Pure `virtual` Classes - Interfaces

So what happens if we declare all our methods as pure virtual? This is what is normally called an *interface* (or an abstract class, although this term is normally reserved for classes that have only some of their methods declared as pure virtual).

In C++ there is one thing you should remember when declaring a pure virtual class - **always define the virtual destructor**. Without the `virtual` destructor, the class does not know that it should go to the virtual method table and call the correct chain of destruction - leading to memory leaks.

Let us for the last time look at our animal example. Our `animal` class is an obvious example of an interface, but so are `bird`, `mammal`, `reptile`, etc. - these have to default behaviour. However, a `parrot` or a `dog` could - they say `squak` or `woof`.

Figure 9.3 provides a diagram of the inheritance hierarchy we just defined. You should try and implement this yourself first before looking at the code below.

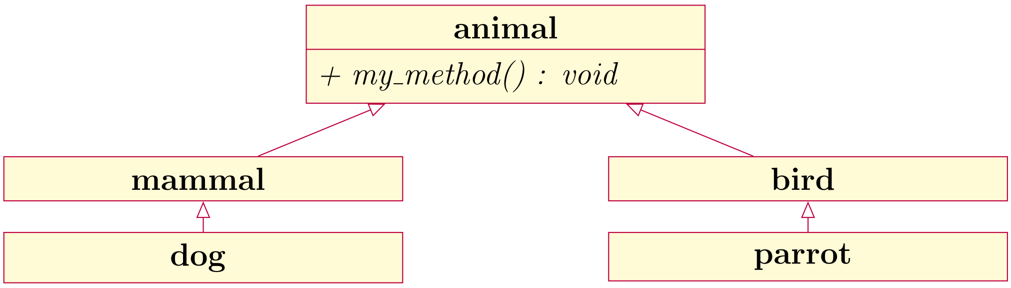


Figure 9.3: Animal Class Diagram

```cpp
#include <iostream>

using namespace std;

class animal
{
public:
  // **** ALWAYS DEFINE THE VIRTUAL DESTRUCTOR ****
  virtual ~animal()
  {
    cout << "Called animal destructor" << endl;
  }

  virtual void speak() = 0;
};

class bird : public animal
```

```
{
public:
  ~bird()
  {
    cout << "Called bird destructor" << endl;
  }

  // Not implementing anything - still an interface
};

class mammal : public animal
{
public:
  ~mammal()
  {
    cout << "Called mammal destructor" << endl;
  }

  // Not implementing anything - still an interface
};

class parrot : public bird
{
public:
  parrot()
  {
    cout << "Called parrot constructor" << endl;
  }

  ~parrot()
  {
    cout << "Called parrot destructor" << endl;
  }

  // Speak implemented here
  void speak() override final
  {
    cout << "Squak!" << endl;
  }
};

class dog : public mammal
{
public:
  dog()
  {
    cout << "Called dog constructor" << endl;
  }

  ~dog()
  {
    cout << "Called dog destructor" << endl;
  }

  // Speak implemented here
  void speak() override final
  {
    cout << "Bark!" << endl;
  }
};
```

```cpp

int main(int argc, char **argv)
{
  // Create a parrot and a dog
  bird *p = new parrot;
  mammal *d = new dog;

  // Tell the animals to speak
  p->speak();
  d->speak();

  // Destroy the animals
  delete p;
  delete d;

  return 0;
}
```

Listing 9.20: Defining Interfaces in C++

Notice that our `bird` and `mammal` classes do not implement `speak`. Therefore these classes are still abstract (they are in fact interfaces). Notice also that we declare our `speak` methods in `dog` and `parrot` as `override final` - these methods will be overridden no further.

You should be able at this stage to determine the output from this application before running it, but just in case it is given below:

```
Called parrot constructor
Called dog constructor
Squak!
Bark!
Called parrot destructor
Called bird destructor
Called animal destructor
Called dog destructor
Called mammal destructor
Called animal destructor
```

Listing 9.21: Output from Interface Application

9.10 Exercises

Return to the Pegasus exercise you undertook at the end of the previous unit and update it to use `virtual` behaviours. Then expand your application to support a Griffin (part lion, part eagle), Harpy (part person, part bird), Centaur (part person, part horse), Mermaid (part person, part fish) and a Sphinx (part human, part lion). Ensure these creatures all have the correct inheritance hierarchy and make the correct sounds when asked.

Unit 10

Data Structures using Pointers

We are now going to look at one of the prime motivators for working with pointers and objects in the manner we have - *data structures.* A data structure is a technique where we store our data in a particular manner to enable simple access in a fast and efficient manner.

Our journey is almost over in the module, so let us review what we have covered so far. The units have been:

1. Working with the compiler and linker to understand how our code is transformed from a high level language to instructions that our machine understands.

2. Looking at how the data we define in our applications is stored in memory.

3. Working with assembly code to understand what our high level instructions look like to the computer.

4. Using headers and separate code files to build libraries and working with the pre-processor.

5. Understanding how we pass variables to functions using pass-by-value, pass-by-reference, and pass-by-pointer.

6. Looking at how we allocate memory on the heap and understanding the difference between the stack and heap.

7. Working with the debugger to test our application and visualising the errors that occur.

8. Introducing objects and working with these to allow us to combine our code and data.

9. Expanding on object orientation with the concepts of virtual behaviour allowing us to override default object implementations in sub-classes.

10. Expanding on object orientation with operator overloads to provide a richer object behaviour.

10.1 Data Structures and the Why of Pointers

Pointers and references are an important concept to understand in programming. Without pointers and references we would be unable to access other areas of memory easily. Pointers and references provide us with a level of indirection (or mapping) from one location to another.

When it comes to data structures this is very important. Up until now most of your understanding of data storage likely focuses on arrays, which are primitive blocks in memory. These are fine, but they aren't very efficient. Consider a collection where you have 1 million records unsorted. How do you find a single record? Search one at a time? What about sorting them? Adding a new record? All these are problems which we face in any application of scale.

So what is a data structure then? Well a data structure is a technique of storing information in memory in a manner that suits the particular application purpose. There are a number of data structures out there. We will only look at a couple - you will be doing a whole module on these later in your studies. Figure 10.1 provides a quick overview of some of the most common data structures.

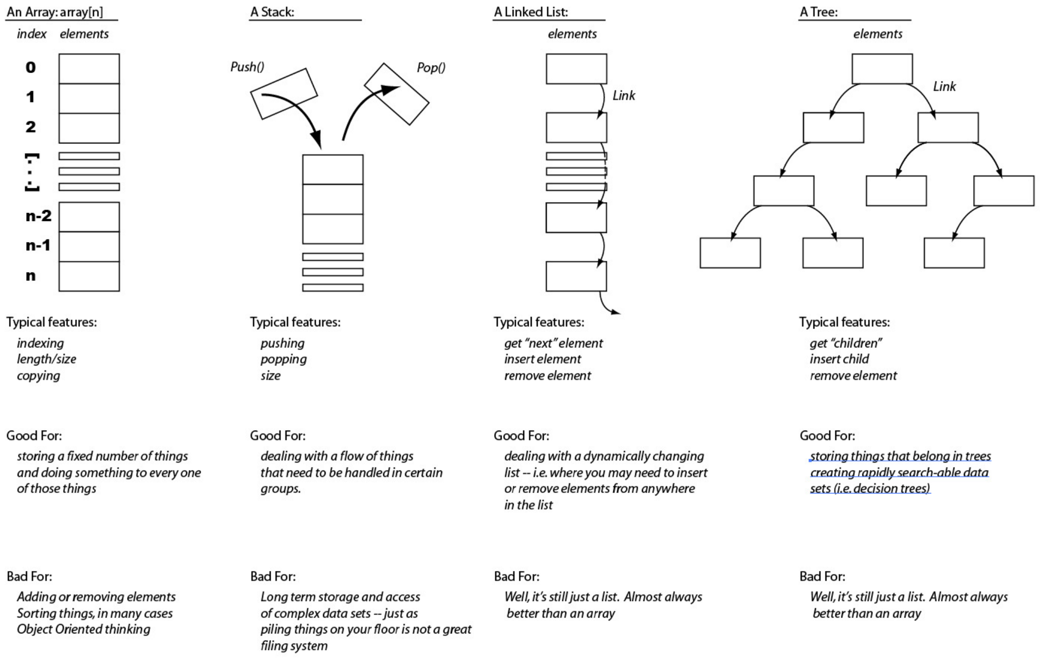

An Array: array[n]
index
elements
0
1
2
[...]
n-2
n-1
n
Typical features:
indexing
length/size
copying
Good For:
storing a fixed number of things and doing something to every one of those things
Bad For:
Adding or removing elements
Sorting things, in many cases
Object Oriented thinking
A Stack:
Push()
Pop()
Typical features:
pushing
popping
size
Good For:
dealing with a flow of things that need to be handled in certain groups.
Bad For:
Long term storage and access of complex data sets -- just as piling things on your floor is not a great filing system
A Linked List:
elements
Link
Typical features:
get "next" element
insert element
remove element
Good For:
dealing with a dynamically changing list -- i.e. where you may need to insert or remove elements from anywhere in the list
Bad For:
Well, it's still just a list. Almost always better than an array
A Tree:
elements
Link
Typical features:
get "children"
insert child
remove element
Good For:
storing things that belong in trees creating rapidly search-able data sets (i.e. decision trees)
Bad For:
Well, it's still just a list. Almost always better than an array

Figure 10.1: Example Data Structures - Retrieved from University of Rochester

10.2 Revisiting Arrays

To start of our discussion on data structures let us look at the simplest one - *the array*. We have looked at arrays a number of times in the module, and you should be familiar with the concept from your previous programming experience. Arrays are essentially a *representation of how data is stored in memory*. That is, an array is a continuous block of memory containing 1 or more elements.

In C and C++ we have seen that this representation in the form of *pointers*. When we declared an array (a block of memory) we could just treat it like a pointer to a location in memory. That is, we can declare a value as `int a[100]` and then access this as a pointer using `int *b = a`.

Another limitation of C and C++ is that to use the syntax `int a[size]` we must know the value of `size` at *compile time*. If `size` is a runtime value (a variable) then the C/C++ compiler will provide a compilation error. This is why we need to use `malloc` and `new`.

Because C and C++ treat arrays as just memory blocks, we don't know how big our array is at runtime. This is a limitation in C and C++ which is not seen in languages such as Java and C#. Therefore, we have to keep track of the size ourselves and pass this around our application.

Let us summarise these four points:

- An array is a continuous block of memory storing 1 or more values
- In C and C++ an array can be treated as a pointer to a data type
- In C and C++ we need to either know the size of the array at compile time, or we have to use `malloc` or `new` to declare the array
- C and C++ do not keep track of the size of the array

The final three points are quite limiting when dealing with C and C++ when you are used to other high level languages which provide an array type that keeps the size for us. This is why we use the `vector` type in C++.

Let us look at a simple example program where we revisit the array type.

```
#include <iostream>

using namespace std;

// A const global size value
const unsigned int SIZE = 1024;

int main(int argc, char **argv)
{
  // Create an array on the stack
  int a[10];
  // Create an array on the stack using SIZE
  int b[SIZE];
  // Create an array on the stack using non-constant value
  // Won't compile
  int x = 200;
  int c[x];
  // Use new to allocate an array on the heap
  int *d = new int[100];
  // Do the same using the non-constant value.
  // Will compile
  int *e = new int[x];

  // Output the sizes of the values
  cout << "sizeof(a) = " << sizeof(a) << endl;
  cout << "sizeof(b) = " << sizeof(b) << endl;
  cout << "sizeof(d) = " << sizeof(d) << endl;
  cout << "sizeof(e) = " << sizeof(e) << endl;

  // Delete the allocated memory
  delete[] d;
  delete[] e;
33 
34   return 0;
35 }
```

Listing 10.1: Revisiting Arrays

There are quite a few areas worth pointing out here.

line 11 - we create an `int` array of 10 members. As this is created on the stack at compile time, we know the size of the data (the print on line 25 will output 40 as the size).

line 13 - `SIZE` is a *compile time* constant. Therefore we can use the value to create an array on the stack, and we know the size of the data (the print on line 26 will output 4096 as the size).

line 17 - *this line will not compile!* `x` is a runtime variable and therefore we cannot create an array in this manner. You will have to comment out this line in your implementation.

line 19 - we create an `int` array of 100 members using `new`. As this is created on the heap at runtime, we do not know the size of the data (the print on line 27 will output the size of `int*` which is 4 bytes (or 8 on a 64-bit machine)).

line 22 - we can use `x` as an array size when we use `new`. The size of this data is also unknown (the print on line 28 will be 4 or 8 bytes).

The output from this application is shown below:

```
sizeof(a) = 40
sizeof(b) = 4096
sizeof(d) = 4
sizeof(e) = 4
```

Listing 10.2: Output from Array Application

10.3 Multidimensional Arrays

We can also declare *multidimensional* arrays. A multidimensional array has more than one dimension (or size). That is, for example, we can declare a multidimensional array as `int x[10][10]`. This means we now have 100 values (10 × 10).

A multidimensional array should be thought of a little differently than a normal array. We need to think about what the line `int x[10][10]` is actually telling us.

- we are storing `int` data
- we have an array storing 10 items (the first `[10]`)
- each of these items in the array stores 10 items (the second `[10]`)

Therefore, we actually have 10 arrays storing 10 `int` values each. Or, in C/C++ terms we have 10 pointers to blocks in memory storing 10 `int` values each. What this means is that we could declare the type of this value as *an array of 10* ***`int*`***. Therefore `int *x[10]` could also be the type. Furthermore, as an array is also a pointer, we could take this further to `int **x`. Figure 10.2 illustrates the general idea.

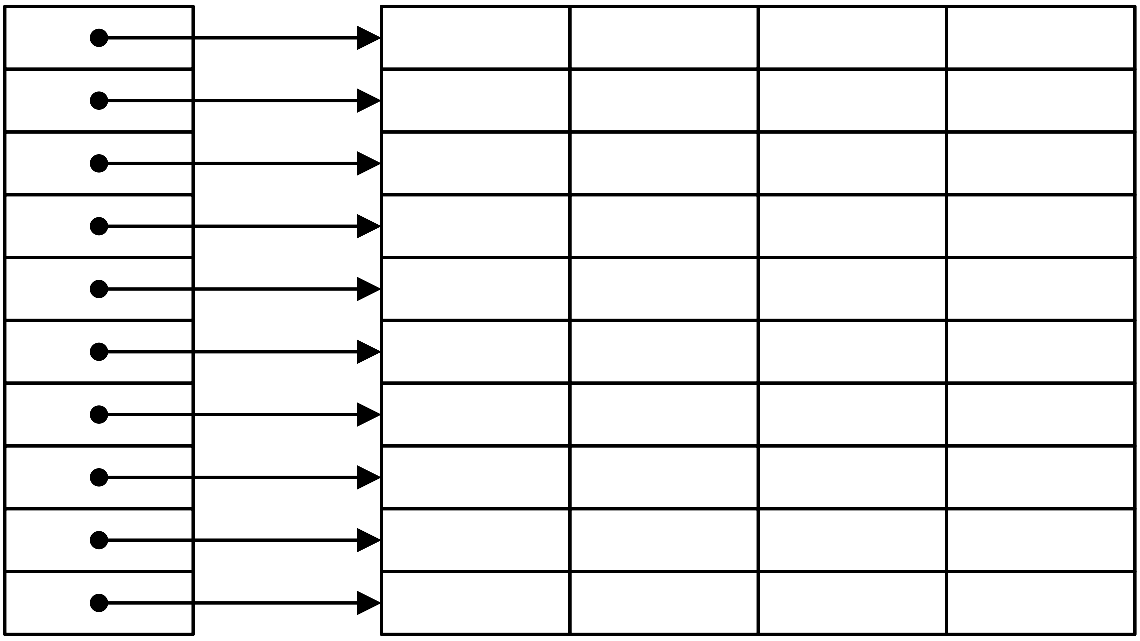


Figure 10.2: Multidimensional Array

Understanding what a multidimensional array is in C/C++ is important for working with them. We are dealing with pointers to memory blocks in all dimensions. Knowing that `int x[size]` is the same type as `int *x`, and `int x[size][size]` is the same as `int *x[size]` and `int **x` is the point of clarity (which takes a lot of thought so *really spend time trying to figure this out in your head*).

So what about a declaration of `int x[10][10][10]`? That is just a `int ***x`. This is now getting into levels of indirection beyond the scope of this module. Let us move onto an example. This is below:

```
#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
  // Declare a multidimensional array on the stack
  int a[10][10];
  // Print the sizeof(a)
  cout << "sizeof(a) = " << sizeof(a) << endl;

  // Set the values of a
  for (int i = 0; i < 10; ++i)
    for (int j = 0; j < 10; ++j)
      a[i][j] = i * j;

  // We can treat the individual columns as arrays
  for (int i = 0; i < 10; ++i)
  {
    int *b = a[i];
    for (int j = 0; j < 10; ++j)
      cout << b[j] << " ";
    cout << endl;
  }

  cout << "****" << endl;

```

```
// Declare a multidimensional array on the heap
int **b = new int*[10];
// Need to allocate all the members individually
for (int i = 0; i < 10; ++i)
  b[i] = new int[10];

// Set the values of b
for (int i = 0; i < 10; ++i)
  for (int j = 0; j < 10; ++j)
    b[i][j] = (i * 10) + j;

// Print using same technique
for (int i = 0; i < 10; ++i)
{
  int *c = b[i];
  for (int j = 0; j < 10; ++j)
    cout << c[j] << " ";
  cout << endl;
}

cout << "****" << endl;

// Delete the multidimensional array - have to delete each part
for (int i = 0; i < 10; ++i)
  delete[] b[i];
delete[] b;

// Let's go mad - 3-dimensional array
int ***c = new int**[10];
for (int i = 0; i < 10; ++i)
{
  c[i] = new int*[10];
  for (int j = 0; j < 10; ++j)
    c[i][j] = new int[10];
}

// Set the values
for (int i = 0; i < 10; ++i)
{
  int **d = c[i];
  for (int j = 0; j < 10; ++j)
  {
    int *e = d[j];
    for (int k = 0; k < 10; ++k)
      e[k] = (i * 100) + (k * 10) + j;
  }
}

// Print the values
for (int i = 0; i < 10; ++i)
{
  for (int j = 0; j < 10; ++j)
  {
    for (int k = 0; k < 10; ++k)
      cout << c[i][j][k] << " ";
    cout << endl;
  }
  cout << "****" << endl;
}

```

```
88    // And now delete everything
89    for (int i = 0; i < 10; ++i)
90    {
91      for (int j = 0; j < 10; ++j)
92        delete[] c[i][j];
93      delete[] c[i];
94    }
95    delete c;
96
97    return 0;
98  }
```

Listing 10.3: Multidimensional Arrays

OK, we have a lot going on in here, so spend the time working through this code.

line 8 - we create a new multidimensional array with fixed compile time size. As we know the size at compile time, when we print it out we will get the expected value (400 - `int` ×10 × 10).

lines 13 to 15 - we iterate through the individual values of the array and set them accordingly using the `a[i][j]` syntax.

lines 18 to 24 - we print the multidimensional array. This is done by printing each array member to a line. This is shown in the output sample below.

line 29 - we declare a new multidimensional array. Notice that we use the type `int**`.

lines 31 and 32 - we create the 10 arrays necessary for `b` on the heap. We then set and print the values as before.

lines 51 and 52 - we have to `delete` each array individually. This is required as we created each array individually.

line 53 - we can `delete b`. As the member arrays have been deleted, we can now do this.

line 56 - we are now declaring an array with 3 dimensions.

lines 57 to 62 - we are allocating the individual arrays for the array. Notice we have to allocate the second dimension as a new `int*` before allocating the third dimension. We then set and print the values as before.

lines 89 to 95 - we now `delete` the 3-dimensional array. This requires us to unwind the allocation one step at a time.

One thing that should become apparent working with an array in this manner is that we don't have uniform sizing. For example:

```
int **x = new int*[10];
x[0] = new int[100];
x[1] = new int[2];
x[2] = new int[12];
// ... etc.
```

Although perfectly legal C++, you will find it difficult to keep track of the sizes of the arrays unless you store them elsewhere. This is why data structures become so important.

```
0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9
0 2 4 6 8 10 12 14 16 18
0 3 6 9 12 15 18 21 24 27
0 4 8 12 16 20 24 28 32 36
0 5 10 15 20 25 30 35 40 45
0 6 12 18 24 30 36 42 48 54
0 7 14 21 28 35 42 49 56 63
0 8 16 24 32 40 48 56 64 72
0 9 18 27 36 45 54 63 72 81
```

Listing 10.4: Sample Output from Multidimensional Array Application

10.3.1 For the Brave - Exercise

As a little exercise to test those who think they really get multidimensional arrays write a function that takes two multidimensional arrays of equal sizing and adds them together. For example:

```c
void add_arrays(int **a, int **b, int **result, int width, int
    height)
{
  // ... other code
  result[x][y] = a[x][y] + b[x][y];
}
```

10.4 Inserting Values into Arrays

So we have arrays to store data. So why do we need anything more? Well, arrays have some limitations that lead to big problems when working with large data sets. The one we will look at is inserting a value into an array. That is, how do we take an array of n values, insert a value v into position k so that everything moves one place down from that point. In other terms, we are doing the following:

$$\begin{aligned}
x'[0] &= x[0] \\
x'[1] &= x[1] \\
&\dots \\
x'[k-1] &= x[k-1] \\
x'[k] &= v \\
x'[k+1] &= x[k] \\
&\dots \\
x'[n-1] &= x[n-2] \\
x'[n] &= x[n-1] \\
x'[n+1] &= x[n]
\end{aligned}$$

Two things become apparent with the above illustration. Firstly, the result (x') is going to be 1 item larger than the starting array. Secondly, after $x[k]$ each value needs to be moved up one place. We can therefore break down the solution into the following steps:

1. Create a new array of $n + 1$ elements

2. Copy 0 to $k - 1$ elements from the starting array to the new array
3. Set value k to v in the new array
4. Copy k to n elements from the starting array to the new array in positions $k + 1$ to $n + 1$
5. Delete the original array

Points 2 and 4 above require us to use the copying features of C and C++.

Copying Data in C++

C++ provides a function called `copy` to allow us to copy data from one collection to another. It actually works on a number of different data types including arrays. The basic structure is as follows:

```
copy(start, end, destination)
```

The values it takes are as follows:

start - the starting point to copy from

end - the end point to copy from

destination - the starting location to copy to

As we are working with arrays, these values are all memory locations. We can use a bit of pointer magic to get these values. If `x` is the array of values (a pointer type):

```
copy(x + 0, x + (k - 1), x' + 0);
x'[k] = v;
copy(x + k, x + n, x' + k + 1);
```

This is a bit of pointer magic. We are telling the compiler to go to, for example, the memory k elements into the array.

C does not have `copy` so we have to perform a memory copy instead. The principle is essentially the same for arrays, but it doesn't work for more complex data types:

```
memcpy(destination, source, size)
```

The values it takes are as follows:

destination - location in memory to copy to

source - location in memory to copy from

size - number of bytes to copy

The `algorithm` Header

The `copy` function comes as part of the `algorithm` header in C++. This is a useful set of functions for working with data and if you continue using C++ in any form you should get to know these functions. You can find out more here.

Now let us look at an example application where we insert a value into an array. This is shown below:

```cpp
#include <iostream>
#include <algorithm>

using namespace std;

// Inserts a value into the array at the given index
void insert(int **array, int &size, int index, int value)
{
  // Create a new array of size + 1
  int *temp = new int[size + 1];
  // Copy 0 to index to the new array
  copy(*array + 0, *array + index, temp + 0);
  // Set temp[index] to value
  temp[index] = value;
  // Copy index to size - 1 to the new array
  copy(*array + index, *array + size, temp + index + 1);
  // Delete array pointed to
  delete[] *array;
  // Set array to temp
  *array = temp;
  // Increment size
  ++size;
  // Values have now been set
}

int main(int argc, char **argv)
{
  int size = 5;
  // Create a new array of 5 values
  int *array = new int[size];
  array[0] = 0;
  array[1] = 1;
  array[2] = 3;
  array[3] = 4;
  array[4] = 5;

  // Print the values
  for (int i = 0; i < size; ++i)
    cout << array[i] << " ";
  cout << endl;

  // Insert 2 into position 2
  insert(&array, size, 2, 2);

  // Print the values
  for (int i = 0; i < size; ++i)
    cout << array[i] << " ";
  cout << endl;

  return 0;
}
```

Listing 10.5: Inserting Values into Arrays

The example output from this code is as follows:

```
0 1 3 4 5
0 1 2 3 4 5
```

Listing 10.6: Output from Array Insertion Application

As you can imagine, removing items from the array is also an extra bit of code. Other functions will take more code. This is where the power of data structures comes in. We can create reusable objects that can manage our data for us. We already used one a few lessons back - *the `vector`* class.

10.5 Revisiting `vector`

We previously used the `vector` type when working with libraries. This was in place of an array to store data in C++. This is effectively what the `vector` is for - storing data in an array like fashion. What it does provide is methods to add, insert and remove values from the `vector`, something that the array doesn't do. In practice, you should prefer the `vector` to the array in C++ in almost all cases.

Other `vector` Methods

The `vector` type provides a number of new methods. Here is a brief list of some of these:

size - gets the number of elements in the `vector`

empty - returns true if the `vector` is empty

push_back - adds an item to the end of the `vector`

pop_back - removes an item from the end of the `vector`

insert - inserts a value into the `vector`

erase - removes a value from the `vector`

clear - empties the `vector`

Inserting Values into a `vector`

The `vector` can insert values in a few different ways, but from our point of view we are interested in the simplest:

```
insert(position, value)
```

The `position` value is interesting as it requires us to use a concept called `iterators`. We won't get into this in this module (again you will get to it later in your studies). However, we can use the same trick as we did with pointers using the `begin` method on the `vector`. For example to insert a value into position 5 we would do the following:

```
vec.insert(vec.begin() + 5, value);
```

Below is an example application using the `vector` type.

```
#include <iostream>
#include <vector>

```

```cpp
using namespace std;

int main(int argc, char **argv)
{
  // Create a new vector
  vector<int> vals;

  // Add some values to the end of the vector
  for (int i = 0; i < 10; ++i)
    vals.push_back(i);

  // Print
  for (int i = 0; i < vals.size(); ++i)
    cout << vals[i] << " ";
  cout << endl;

  // Insert a value at index 4
  vals.insert(vals.begin() + 4, 100);

  // Print
  for (int i = 0; i < vals.size(); ++i)
    cout << vals[i] << " ";
  cout << endl;

  // Erase value at index 5
  vals.erase(vals.begin() + 5);

  // Print
  for (int i = 0; i < vals.size(); ++i)
    cout << vals[i] << " ";
  cout << endl;

  // Clear the vector
  vals.clear();

  // Output size
  cout << "Size of vals = " << vals.size() << endl;

  return 0;
}
```

Listing 10.7: Revisiting `vector`

The output from this example is below:

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 100 4 5 6 7 8 9
0 1 2 3 100 5 6 7 8 9
Size of vals = 0
```

Listing 10.8: Output from `vector` Application

We will return to `vector` one last time towards the end of this unit.

10.6 Creating Data Structures Using Pointers

So we have arrays which are good unless we want to insert or remove values, and we have `vector` which takes care of these limitations for us. So we are done right?

Well no. We have so far looked at storing data in continuous blocks of memory, but these are terrible for sorting, storing in order, and other issues. And although we have overcome the limitation of insertion and removing data, these are very slow methods of doing this. So how about other data structures.

Before moving onto these ideas, let us refresh our knowledge of working with pointers. Pointers allow us to overcome the limitations of storing data in continuous blocks by allowing us to jump to other locations in memory as required.

So lets look at an example. Let us rebuild our student application using some pointers. This code is shown below, and the object structure shown in Figure 10.3.

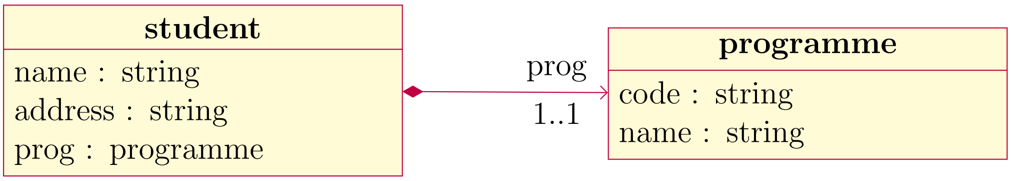


Figure 10.3: Class Diagram for Student Application

```
#include <iostream>
#include <string>
#include <memory>

using namespace std;

// Programme info
struct programme
{
  string code;
  string name;

  programme(const string &c, const string &n)
  {
    code = c;
    name = n;
  }
};

// Student info
struct student
{
  string name;
  string address;
  // Pointer to a programme
  // Shared data
  shared_ptr<programme> prog;

  student(const string &n, const string &a, shared_ptr<programme> p)
  {
    name = n;
    address = a;
    prog = p;
  }
};

void print_student(const student &s)
{
  cout << s.name << endl;
  cout << s.address << endl;
  cout << s.prog->code << " " << s.prog->name << endl;
42    cout << endl;
43  }
44
45  int main(int argc, char **argv)
46  {
47    // Create two programmes
48    auto games = make_shared<programme>("56209BH", "BSc (Hons) Games
          Development");
49    auto se = make_shared<programme>("56178BI", "MEng Software
          Engineering");
50
51    // Create four students
52    student stud1("Kevin Chalmers", "School of Computing", games);
53    student stud2("Lu Fan", "School of Computing", se);
54    student stud3("Neil Urquhart", "School of Computing", se);
55    student stud4("Ben Kenwright", "School of Computing", games);
56
57    // Print the students
58    print_student(stud1);
59    print_student(stud2);
60    print_student(stud3);
61    print_student(stud4);
62
63    return 0;
64  }
```

Listing 10.9: Creating Data Structures Using Pointers

On line 27 we are now declaring a shared_ptr to a programme for each student. This means that we can reuse our programme objects. This we do on lines 52 to 55 when we actually create some student objects. The output from this application is below:

```
Kevin Chalmers
School of Computing
56209BH BSc (Hons) Games Development

Lu Fan
School of Computing
56178BI MEng Software Engineering

Neil Urquhart
School of Computing
56178BI MEng Software Engineering

Ben Kenwright
School of Computing
56209BH BSc (Hons) Games Development
```

Listing 10.10: Output from Data Structures Application

The idea we are trying to get across here is that we can use a pointer to join data together. The student object points to a programme object. We effectively jump to the location in memory where the programme information is stored. We can expand this principle for storing data.

10.7 Creating a Linked List

Let us build an example data structure - *a linked list*. This data structure is used to store data in a list fashion. That is, we have a first item, then a second, etc. Each item in the list points to the next item. If the next item is empty (i.e. a nullptr) then we have reached the end of the list. Figure 10.4 illustrates the basic structure of a linked list.

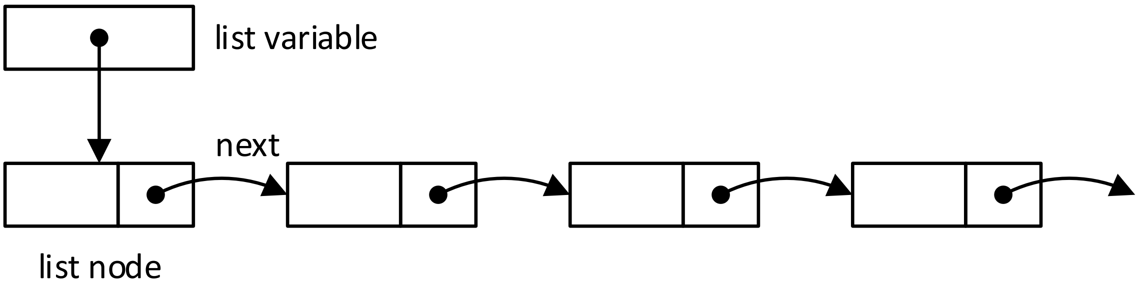


Figure 10.4: Linked List

What is a Linked List?

So why do we have linked lists? Surely this is just like a `vector` or array but with the next item located somewhere else in memory rather than being continuous. Well a linked list has one advantage - *it is very fast for inserting and removing values!*

Think about what we have to do when inserting a value into an array or `vector` (which is just an array underneath). There is a lot of data copying. For large data sizes, this becomes almost the entire time spent on the operation of insertion, copying the data.

For a linked list, we just go to the location where the data is stored, create a new value to point to, and set the new value's next to the previous next value. It is that easy.

For a linked list to work we need to introduce the idea of a data node. We already looked at this when we did the binary tree case study.

The list node needs to point to the next member and also maintain its own value. In our application we will just store `int` data. Our `list_node` type is shown below:

```
 1 struct list_node
 2 {
 3   // Value stored on this node
 4   int value = 0;
 5   // Pointer to the next node
 6   shared_ptr<list_node> next = nullptr;
 7
 8   list_node(int v)
 9   {
10     value = v;
11   }
12 };
```

It is a very simple data type, but underpins a powerful concept. Line 6 contains a pointer to another `list_node`. We therefore have a recursive data type. We can keep accessing the next value until we hit a `nullptr`.

The recursive nature of the data type underpins how we perform operations on the linked list. For example, let us consider how we add a value to the end of the linked list. We can think of it as follows:

1. Get first node

2. If node is `nullptr` create a new node and set to first
3. else get next node
4. If node is `nullptr` create a new node and set to next
5. else get next node
6. If node is `nullptr` create a new node and set to next
7. else get next node
8. ... etc.

Obviously we cannot keep writing infinite if statements for all the different configurations we might meet. We could write this as a `while` loop as shown in Algorithm 9.

Algorithm 9 Appending to a Linked List Using a While Loop

```
procedure APPEND($list$, $value$)
    $next \leftarrow list$
    while $next \neq nullptr$ do
        $next \leftarrow next.next$
    $next \leftarrow$ LIST_NODE($value$)
```

However, we will use a recursive function (since you need to understand these). This is shown in Algorithm 10.

Algorithm 10 Appending to a Linked List Using a Recursive Function

```
procedure APPEND($list$, $value$)
    if $list = nullptr$ then
        $list.next \leftarrow$ LIST_NODE($value$)
    else
        APPEND($list.next$, $value$)
```

Our example application is below. *Ensure you step through this application and understand what it is doing!*

```cpp
#include <iostream>
#include <memory>

using namespace std;

// A node in the linked list
struct list_node
{
  // Value stored on this node
  int value = 0;
  // Pointer to the next node
  shared_ptr<list_node> next = nullptr;

  list_node(int v)
  {
    value = v;
  }
};
```

```

// Add a value to the end of the list
void append(shared_ptr<list_node> list, int value)
{
  // Check if we are the last node
  if (list->next == nullptr)
  {
    // Add a new node to next
    list->next = make_shared<list_node>(value);
  }
  else
  {
    // Not the end of the list.
    // Try and append to the next node
    append(list->next, value);
  }
}

// Print the list
void print(shared_ptr<list_node> list)
{
  // Print value
  cout << list->value << " ";
  // Check if this is the end of the list
  if (list->next == nullptr)
  {
    cout << "END" << endl;
  }
  else
  {
    // Otherwise print next node
    print(list->next);
  }
}

int main(int argc, char **argv)
{
  // Create a new list
  auto list = make_shared<list_node>(0);

  // Print the list
  print(list);

  // Append 10 values
  for (auto i = 0; i < 10; ++i)
    append(list, i + 1);

  // Print the list
  print(list);

  return 0;
}
```

Listing 10.11: Linked List Application

Note as well that our `print` statement is also recursive. The output from this application is below.

```
0 END
0 1 2 3 4 5 6 7 8 9 10 END
```

Listing 10.12: Output from Linked List Application

10.8 Library Data Structures

We have really only looked at a brief idea of how to build data structures. The main point of this lesson is to actually introduce you to the library data structures.

Each modern programming language or runtime has a collection of data structures that it supports. You will find a great deal of commonality between the supported data structures. We will only look at a few of the most common here. Table 10.1 illustrates the ones we will look at.

Name	Example	Description
`vector`	`vector<int>`	Stores data in a continuous manner in memory. Operates much the same as an array but can be modified at runtime.
`map`	`map<int, string`	Stores data in a key-value pair manner, also known as associated data storage. We use the key to look up the value we are interested in
`list`	`list<int>`	Stores data in a continuous manner, but not continuous in memory. Allows quick insert and remove at the expensive of traversing the data
`stack`	`stack<int>`	Stores data in a first-in, last-out (FILO) manner. Data is pushed onto the top of the stack and popped from the top also.
`queue`	`queue<int>`	Stores data in a first-in, first-out (FIFO) manner. Data is pushed onto the end of the queue and popped from the front
`set`	`set<int>`	Stores data so that no duplicate values are stored. That is, if 5 is already in the `set`, any attempt to add 5 again is ignored

Table 10.1: Common Data Structures in C++

10.8.1 vector

We have covered the `vector` type a number of times already. The example below only adds one new idea.

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

int main(int argc, char **argv)
{
  // Create a vector of string with 10 values
  vector<string> vec(10);

  // Print out size
  cout << "Size of vec = " << vec.size() << endl;

  // Prompt for and set the ten values
  for (int i = 0; i < vec.size(); ++i)
18   {
19     string s;
20     cout << "Enter name: ";
21     cin >> s;
22     vec[i] = s;
23   }
24 
25   // Print the names
26   for (int i = 0; i < vec.size(); ++i)
27     cout << vec[i] << endl;
28 
29   // Sort the names
30   cout << "Sorting the entries" << endl;
31   sort(vec.begin(), vec.end());
32 
33   // Remove the first five values
34   cout << "Erasing 5 entries" << endl;
35   vec.erase(vec.begin(), vec.begin() + 5);
36 
37   // Print out size
38   cout <<"Size of vec = " << vec.size() << endl;
39 
40   // Print the names
41   for (int i = 0; i < vec.size(); ++i)
42     cout << vec[i] << endl;
43 
44   return 0;
45 }
```

Listing 10.13: Using `vector`

On line 31 we use the `sort` function. This function is part of the standard library and can be used to sort a collection of data. The type of sorting it performs is *very efficient*. Much more so than the bubble sort approach we took previously.

Using `sort`
The `sort` function takes the following structure:

`sort(begin, end)`

It just needs the starting point and end point to sort. Therefore, we could sort different parts of the collection if we wanted to.

An example run of this application is shown below:

```
Size of vec = 10
Enter name: Kevin
Enter name: Neil
Enter name: Simon
Enter name: Ben
Enter name: Lu
Enter name: Andrew
Enter name: Frank
Enter name: Tom
Enter name: Rob
Enter name: Bob
Kevin
Neil
Simon
Ben
Lu
Andrew
```

```
Frank
Tom
Rob
Bob
Sorting the entries
Erasing 5 entries
Size of vec = 5
Lu
Neil
Rob
Simon
Tom
```

Listing 10.14: Output from Using `vector`

10.8.2 Ranged `for` Loop

Before moving onto the `map` data structure let us introduce the *ranged for loop* in C++. This `for` loop takes a collection of data and provides us with access to its individual members. This allows us to write simpler code than if we were using the standard `for (int i = 0; i < data.size(); ++i)`

Ranged `for` Loop in C++
To write a ranged `for` loop in C++ we use the following syntax:

```
for (auto &v :  data)
```

We use `auto` so that we don't have to worry about the data value we are accessing - the compiler will work this out for us. Notice also that we get a reference value (`&v`). This is so we don't copy the data out of the collection, and we can also modify the value if we want to. The use of the colon (:) is tell the compiler which data collection we wish to access.

As an example, let us rewrite our `vector` example using a ranged `for` loop. This is shown below.

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

int main(int argc, char **argv)
{
  // Create a vector of string with 10 values
  vector<string> vec(10);

  // Print out size
  cout << "Size of vec = " << vec.size() << endl;

  // Prompt for and set the ten values
  // Use ranged for
  for (auto &v : vec)
  {
    cout << "Enter name: ";
    cin >> v;
  }
23
24   // Print the names
25   // Use ranged for
26   for (auto &v : vec)
27     cout << v << endl;
28
29   // Sort the names
30   cout << "Sorting the entries" << endl;
31   sort(vec.begin(), vec.end());
32
33   // Remove the first five values
34   cout << "Erasing 5 entries" << endl;
35   vec.erase(vec.begin(), vec.begin() + 5);
36
37   // Print out size
38   cout <<"Size of vec = " << vec.size() << endl;
39
40   // Print the names
41   // Use ranged for
42   for (auto &v : vec)
43     cout << v << endl;
44
45   return 0;
46 }
```

Listing 10.15: Using Ranged `for` with `vector`

Notice what we do on lines 18 to 21. We are inputting the values directly into the `v` value. This works because we have a reference to the value in the `vector`. We can happily set values because we are referencing them.

You will find a form of ranged `for` loop in most modern languages (called `foreach` in C#). They do make your life easier.

10.8.3 `map`

Our first new data type we will look at is the `map`. A `map` works on the principle of *association data*. That is, we *map* a key to a data value.

A `map` can be considered like a table:

key	**value**
1	Fred
2	Bob
999	Frank
...	...

Or we can use the map-to notation used in mathematics:

$$
\begin{aligned}
1 &\mapsto Fred \\
2 &\mapsto Bob \\
999 &\mapsto Frank \\
\cdots &\mapsto \cdots
\end{aligned}
$$

What this means in practice is that we can have lookup storage based on any type we want. For example, it is perfectly acceptable to write the following code:

```
// Store names in map against age
map<string, unsigned int> ages;
ages["Kevin"] = 36;
ages["Bob"] = 41;
ages["Mary"] = 24;
\\ ... etc.
```

The underlying implementation of the `map` may be optimised to support faster lookup or storage. That is outside our discussion. All we are interested in is the ability to use a key look up.

Working with Association Data in C++

As we have already seen, we can add data to a `map` using square brackets as we would in an array:

```
m[key] = value;
```

This syntax also works to access existing members in the `map`:

```
value = m[key];
```

Each value stored in the `map` can be accessed as an *entry*. This has two values - `first` (the key) and `second` (the value).

Our example application is below.

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

int main(int argc, char **argv)
{
  // Create a new map
  map<unsigned int, string> students;

  // Prompt for and read in student data
  for (int i = 0; i < 10; ++i)
  {
    unsigned int matric;
    string name;
    cout << "matric: ";
    cin >> matric;
    cout << "name: ";
    cin >> name;
    students[matric] = name;
  }

  // Print out the details
  // Use ranged for
  for (auto &entry : students)
    cout << entry.first << " " << entry.second << endl;

  // Overwrite some details
  for (int i = 0; i < 3; ++i)
  {
```

```
32     unsigned int matric;
33     string name;
34     cout << "matric to change: ";
35     cin >> matric;
36     cout << "new name: ";
37     cin >> name;
38     students[matric] = name;
39   }
40 
41   // Print out the details
42   // Use ranged for
43   for (auto &entry : students)
44     cout << entry.first << " " << entry.second << endl;
45 
46   return 0;
47 }
```

Listing 10.16: Using `map`

On line 21 we add values to the `map` using the standard array syntax. This is the standard method of adding values to a `map` although there are others.

On line 27 we are using the `first` and `second` values after accessing the individual `map` eateries using a ranged `for` loop.

Also of not is how we can overwrite data - we just use the indexer syntax again. If the value already exists in the `map` we will overwrite it.

An example output from this application is shown below:

```
matric: 0
name: Kevin
matric: 9
name: Neil
matric: 4
name: Simon
matric: 2
name: Ben
matric: 3
name: Andrew
matric: 1
name: Lu
matric: 8
name: Frank
matric: 7
name: Tom
matric: 6
name: Rob
matric: 5
name: Bob
0 Kevin
1 Lu
2 Ben
3 Andrew
4 Simon
5 Bob
6 Rob
7 Tom
8 Frank
9 Neil
matric to change: 4
new name: Sally
matric to change: 8
new name: Ken
matric to change: 1
new name: Emma
0 Kevin
1 Emma
2 Ben
3 Andrew
4 Sally
5 Bob
```

```
6 Rob
7 Tom
8 Ken
9 Neil
```

Listing 10.17: Output from `map` Application

10.8.4 `list`

Our next data structure is the `list`. You should be familiar with this data type from our examination of linked list. There is really only one big difference from our original linked list and the C++ `list`. This is that `list` is *doubly linked*. This means a single node points both to the next node and the previous node. This has advantages that we will not discuss here.

Working with `list`

`list` comes with a number of methods similar to `vector`. It does also add the `push_front` method which adds a value to the start of the list. Otherwise the external behaviour is fairly similar.

Our example application is below.

```
#include <iostream>
#include <list>
#include <string>
#include <algorithm>

using namespace std;

int main(int argc, char **argv)
{
  // Create a list of string
  list<string> l;

  // Prompt for and push ten values
  for (int i = 0; i < 10; ++i)
  {
    string name;
    cout << "Enter name: ";
    cin >> name;
    l.push_back(name);
  }

  // Print the names
  for (auto &val : l)
    cout << val << endl;

  // Prompt for and push five values to the front
  for (int i = 0; i < 5; ++i)
  {
    string name;
    cout << "Enter name: ";
    cin >> name;
    l.push_front(name);
  }

  // Print the names
  for (auto &val : l)
    cout << val << endl;

  return 0;
}
```

Listing 10.18: Using `list`

An example output is below:

```
Enter name: Kevin
Enter name: Neil
Enter name: Simon
Enter name: Rob
Enter name: Lu
Enter name: Andrew
Enter name: Bob
Enter name: Frank
Enter name: Tom
Enter name: Ben
Kevin
Neil
Simon
Rob
Lu
Andrew
Bob
Frank
Tom
Ben
Enter name: Emma
Enter name: John
Enter name: Sally
Enter name: Ken
Enter name: Colin
Colin
Ken
Sally
John
Emma
Kevin
Neil
Simon
Rob
Lu
Andrew
Bob
Frank
Tom
Ben
```

Listing 10.19: Output from `list` Application

10.8.5 `stack`

Our next data structure is the `stack`. We have already discussed what a stack is when we looked at inline assembly code. The C++ `stack` provides the same features (*push* and *pop*) except there is no limit on the size, and we can store different types.

In C++ terms, there is actually no `stack` data structure as such. The `stack` type is a modifier on top of another sequential data store (for example `vector` or `list`) that we use to provide stack-like behaviour. We will look at this presently.

Working with `stack`

There are three functions of interest when working with the C++ `stack`:

push - pushes (adds) an item onto the top of the stack

pop - pops (removes) an item from the top of the stack. This item is not returned by `pop` and is therefore lost

top - gets (but does not remove) the item at the top of the stack

Below is an example application for working with the C++ `stack` type.

```
 1 #include <iostream>
 2 #include <vector>
 3 #include <stack>
 4 #include <string>
 5
 6 using namespace std;
 7
 8 int main(int argc, char **argv)
 9 {
10   // Create a stack using a vector
11   stack<string, vector<string>> st;
12
13   // Prompt for and read in names until 0 is entered
14   while (true)
15   {
16     string name;
17     cout << "enter name: ";
18     cin >> name;
19     if (name == "0")
20       break;
21     else
22       st.push(name);
23   }
24
25   // Print out size
26   cout << "Number of names entered: " << st.size() << endl;
27
28   // Print all the names pushed on the stack
29   while (!st.empty())
30   {
31     // Get top value
32     auto val = st.top();
33     cout << val << endl;
34     // Pop the value off the stack
35     st.pop();
36   }
37
38   return 0;
39 }
```

Listing 10.20: Using `stack`

Notice on line 11 we create the `stack` by specifying the type of internal data store (here we use `vector`). Otherwise we are just creating a data store as standard.

On line 22 we add values onto the `stack` using the `push` method.

Lines 29 to 35 show how we go about printing the `stack`. We keep looping until the `stack` is empty, getting the top value, printing it, and then removing it from the `stack` using `pop`.

An example output from this application is below:

```
enter name: Kevin
enter name: Neil
enter name: Simon
enter name: Ben
enter name: Lu
enter name: Andrew
enter name: Frank
enter name: Tom
enter name: Rob
enter name: Bob
enter name: 0
Number of names entered: 10
Bob
Rob
Tom
Frank
Andrew
Lu
Ben
Simon
Neil
Kevin
```

Listing 10.21: Output from `stack` Application

10.8.6 queue

The `queue` in C++ is similar to `stack` in that it requires another sequential data store (`vector`, `list`) to operate. Its main difference is that we don't remove items from the top, but the bottom. We call the front of the queue the `head` and the back the `tail`. Otherwise, a `queue` operates much like a `stack`.

Working with `queue`
We are interested in three methods again:

push - adds a value to the end of the `queue`

pop - removes a value from the start of the `queue`. This value is *not* returned

front - gets (but does not remove) the item from the start of the `queue`

Our example application is below. It is much the same as our `stack` application.

```
#include <iostream>
#include <list>
#include <queue>
#include <string>

using namespace std;

int main(int argc, char **argv)
{
  // Create a queue using a list
  queue<string, list<string>> q;

  // Prompt for and read in names until 0 is entered
  while (true)
  {
    string name;
    cout << "enter name: ";
    cin >> name;
    if (name == "0")
      break;
    else
      q.push(name);
  }

  // Print out size
  cout << "Number of names entered: " << q.size() << endl;

  // Print all the names pushed on the stack
  while (!q.empty())
  {
    // Get top value
    auto val = q.front();
    cout << val << endl;
    // Pop the value off the stack
    q.pop();
  }

  return 0;
}
```

Listing 10.22: Using `queue`

The main difference to our `stack` application is that we are using a `list` as our underlying data store. This is just for illustration as there is no optimisation need for this. The output (below) is different though due to the different method of accessing the data.

```
enter name: Kevin
enter name: Neil
enter name: Simon
enter name: Ben
enter name: Lu
enter name: Andrew
enter name: Frank
enter name: Tom
enter name: Rob
enter name: Bob
enter name: 0
Number of names entered: 10
Kevin
Neil
Simon
Ben
Lu
Andrew
Frank
Tom
Rob
Bob
```

Listing 10.23: Output from `queue` Application

10.8.7 `set`

Our final data structure we will look at is the `set`. This data store allows us to ensure a value is only ever stored once. Apart from that, there is no sense of order. The store may be optimised for inserting, removing and finding values, or it might not be. Really the point of the `set` is to avoid adding a value more than once to a data store.

Working with `set`

The only method we will look at using `set` is `emplace`. `emplace` adds a value to the `set`. However, it's return type has two values - `first` and `second` like an entry in the table. So what are these values?

- if the value being emplaced is not in the `set` then its new position (as an iterator - just ignore this) is put in the `first` value and `second` is set to `true`
- if the value being emplaced is in the `set` then the existing position is put in `first` and `second` is set to `false`

Our test application is below:

```cpp
#include <iostream>
#include <string>
#include <set>

using namespace std;

int main(int argc, char **argv)
{
  // Create a set
  set<string> s;

  // Prompt for and read in names until 0 is entered
  while (true)
  {
    string name;
    cout << "enter name: ";
    cin >> name;
    if (name == "0")
      break;
    else
      s.emplace(name);
  }

  // Print out size
  cout << "Number of names entered: " << s.size() << endl;

  // Print the values in the set
  for (auto &val : s)
    cout << val << endl;

  // Now try and enter a value already in the set
  string name;
  cout << "enter existing name: ";
  cin >> name;
  auto success = s.emplace(name);
  if (!success.second)
    cout << "Yes, " << name << " already exists" << endl;

  return 0;
}
```

Listing 10.24: Using `set`

You should be able to figure out what is happening in this application based on the code. An example output is below:

```
enter name: Kevin
enter name: Neil
enter name: Simon
enter name: Ben
enter name: Lu
enter name: Andrew
enter name: Frank
enter name: Tom
enter name: Rob
enter name: Bob
enter name: 0
Number of names entered: 10
Andrew
Ben
Bob
Frank
Kevin
Lu
Neil
Rob
Simon
Tom
enter existing name: Lu
Yes, Lu already exists
```

Listing 10.25: Output from `set` Application

10.9 For the Brave - Combining Data Structures: `map` of `vector`

So far our look at data structures has focused on using them individually. However, this is the simplest view. Most of the time we have data that is more complicated than a simple single data structure. Therefore, we combine one or more data structures together.

> **Combining Data Structures**
>
> As you may have noticed, our data structures all declare their type within the angled brackets. For example, to declare a `vector` or `int` we used `vector<int>`.
>
> So what if we wanted to declare a `vector` that stored other `vector` objects which contained `int`. Then this just `vector<vector<int>>`. How about a `set` of these? Then that is `set<vector<vector<int>>>`. And so on.
>
> The main point to think about is how your data is stored. This requires you to understand data - something you will do next session in Databases. You should reflect on the areas we cover here in that module to further understand the concepts.

As an example application consider the following. Here we use a `map` of `vector` to store friendship information, much like you would have on a social media site.

```cpp
#include <iostream>
#include <string>
#include <map>
#include <vector>

using namespace std;

```

```
int main(int argc, char **argv)
{
  // Create a map of vectors
  map<string, vector<string>> friendships;

  while (true)
  {
    // Person name
    string name;
    // Vector of friends
    vector<string> friends;
    // Prompt for a name
    cout << "name: ";
    cin >> name;
    // Check if name is 0
    if (name == "0")
      break;
    else
    {
      while (true)
      {
        // Prompt for a friend name
        string friend_name;
        cout << "friend name: ";
        cin >> friend_name;
        // Check if friend_name is 0
        if (friend_name == "0")
          break;
        else
          friends.push_back(friend_name);
      }
      // Add name and vector of friends to map
      friendships[name] = friends;
    }
  }

  // Print out all the friendships
  for (auto &entry : friendships)
  {
    // Print name
    cout << entry.first << ":" << endl;
    // Print out all the friends
    for (auto &friend_name : entry.second)
      cout << "\t" << friend_name << endl;
  }

  return 0;
}
```

Listing 10.26: Combining map and vector

An example output from this application is below:

```
name: Kevin
friend name: Ben
friend name: Neil
friend name: Rob
friend name: 0
name: Ben
friend name: Lu
friend name: Neil
friend name: Andrew
friend name: 0
name: Rob
```

```
friend name: Neil
friend name: Andrew
friend name: 0
name: Lu
friend name: Kevin
friend name: Ben
friend name: Frank
friend name: Tom
friend name: 0
name: 0
Ben:
        Lu
        Neil
        Andrew
Kevin:
        Ben
        Neil
        Rob
Lu:
        Kevin
        Ben
        Frank
        Tom
Rob:
        Neil
        Andrew
```

Listing 10.27: Output from `map` of `vector` Application

10.10 For the Brave - C++ array Type

To end our look at data structures, let us return to the array. C++ actually does provide an `array` type which can be used with a number of the algorithms discussed in this unit. It is still of fixed size at compile time, but it does keep its size. Currently it is an interesting aside, although the new C++14 standard may make this more useful with the introduction of runtime sizing.

The application below shows how we can declare and use the `array` type.

```
#include <iostream>
#include <array>

using namespace std;

// Size of array must be known at compile time
const unsigned int SIZE = 1024;

int main(int argc, char **argv)
{
  // Declare an array
  array<int, SIZE> arr;

  // Set values in the array
  for (int i = 0; i < SIZE; ++i)
    arr[i] = 1024 - i;

  // Print all the values in the array
  // Use ranged for
  for (auto &val : arr)
    cout << val << endl;

  return 0;
}
```

Listing 10.28: Using the `array` Type

Line 12 shows how we create an `array`. It requires the *emph* (here `int`) and the *size* (here we use `SIZE` which is compile time fixed to 1024).

10.11 Exercises

At this stage you should be working on the final coursework assessment. The final assessment does include data structures and therefore is a suitable exercise alongside this unit.

Unit 11

Coding Standards for Quality and Security

This final unit will explore some of the real problems that can be faced when writing bad code. Bad code is one of the biggest costs to a company. You may even have heard the term technical debt. This is the idea that your easy solution now has a cost in the future – either from rework or other costs. We will look at some examples of bad code and where you can explore to find more.

Let us do one final review of the content we have covered.

1. We now know how the compiler and linker operate to produce code that the machine understands. Knowing this allows us to produce code that is of good quality. Human readable code is important.
2. We also understand how data is stored in memory. This is crucial, as it leads us to know how to exploit the system.
3. We examined how our high-level code relates to assembly code. Again, knowing this allows us to understand how the machine can be exploited.
4. We saw how to break our code up into separate units via headers. Good code quality comes form composability.
5. We now understand how variables can be passed to functions. This is another important aspect where we can understand code exploits.
6. We have examined memory allocation. Poor memory management can lead to numerous problems and vulnerabilities.
7. We worked with the debugger to test our applications. The debugger allows us to explore potential problems in our code which can lead to vulnerabilities.
8. We have introduced object-orientation and applied its techniques to build better code.
9. We expanded our knowledge of object-orientation with the concepts of virtual behaviour. This allows us to write reusable code interfaces.
10. We expanded object-orientation with operator overloads. This allows even better object interfaces.
11. We examined some data structures and how they can be used to store data. This is a useful set of types to allow simple working with data.

11.1 Why Bad Code is Bad

Admit it. You've written some bad code during this module. For whatever reason, there is a bit of code that you hacked together and ended up just leaving alone. There are many reasons you may have done this:

1. You didn't understand the code being asked for and you cobbled together something that seems to work.

2. You have been unsure about what to do, or a bit lazy, and you searched Stack Overflow for an answer and just pasted it in hoping it would be OK.

3. You left your coursework submission too late and have ended up hacking something to meet the deadline. This is a common reason in industry also.

4. Something other reason just to get by.

The problem is that your code still exists. If you had written it for a piece of production software you would have a big problem. How often have you had a program fail and you have cursed it. What about lost data or work? Maybe even you have had a device die for no reason except what seems to be bad software. We have all been there.

You are training to become a programmer. If you are willing to complain about bad code you should be willing to do something about it by writing good code. It is your duty to the rest of us! Spending that time on actually writing good solutions to problems will make you a better programmer overall as your skills and craft will increase.

You're code is never just for you. You write code that others can use, or will work with to maintain a system. One of the reasons we assess your code quality is that you need to get into good habits now. Developing bad habits will just give you more work later. And just wait until you have to do group work.

And for those of you who use Stack Overflow and do not learn the solution – we know! This will not serve you in the coming years as you struggle to understand advanced techniques. The point is, Stack Overflow is great **IF** you learn the solution and integrate it into your code. If you just copy-and-paste that is all you have learned: to copy-and-paste. The easy path is not going to serve you in the long-term.

So writing good code is a duty, and you have to spend the time learning your craft. Bad code leads to vulnerabilities and inefficiencies that will be problematic later. It also means that you enforce bad habits which take time to overcome. And you are not the special one who this idea does not apply to. Every programmer (including me) should constantly practice to improve their overall coding capability.

11.2 Some Dangerous Code Examples

The key message of this unit is that code can be dangerous. We will look at a number of different reasons how this can be. **Be warned** – these examples are simple but many applications still suffer from the problems illustrated. Our aim here is to show what can happen when you write bad code. But first, we are going to review how the stack works. We will be exploiting this behaviour in our dangerous code examples.

11.2.1 Stack Behaviour

Let us review once again how the stack works. Remember that when you declare local values they are added to the stack in the order that you placed them. For example, if we use the following code example:

```c
#include <stdlib.h>

int main(int argc, char **argv)
{
  int x;
  int y;
  char str[8];
  return 0;
}
```

Listing 11.1: Values on the Stack

Our stack would look like the one shown in Figure 11.1. Note that the stack fills from the top, not the bottom. This is what happens in memory. It allows the stack and heap to grow towards each other.

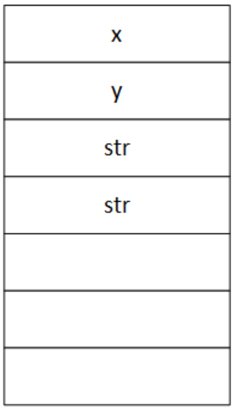


Figure 11.1: Example Stack

The problem is that we can point directly into the stack – we just need a memory address – and overwrite values. This is especially problematic when using operations that are unbounded. An unbounded operation is one that will just copy as much data as possible from a given memory address or to a given memory address. This will lead to accessing unwanted data or overwriting existing data. We discussed this idea when working with character arrays that were not null-terminated. So how dangerous can this be?

11.2.2 Overwriting Data

The following code sample – `password.c` – is an example of what will happen when you overwrite data. We are using the dangerous `gets` operation to read in user input. `gets` is unbounded and will simply read in user input into memory until a new line is detected.

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

#define PASSWORD "password"
```

```
 6
 7 int main(int argc, char **argv)
 8 {
 9   // String to read user input into
10   char user_input[8];
11   // Flag to indicate is login was successful.
12   int success = 0;
13
14   printf("Enter password: ");
15   // **** DANGER ****
16   // Do not use gets
17   gets(user_input);
18
19   // If password is correct grant root access
20   if (strcmp(user_input, PASSWORD) == 0)
21   {
22     printf("Login successful\n");
23     success = 1;
24   }
25   else
26   {
27     printf("Login failed\n");
28   }
29
30   // Check if root access given
31   if (success)
32   {
33     printf("Root access granted\n");
34   }
35
36   return 0;
37 }
```

Listing 11.2: Dangerous Code Example 1 – `password.c`

Looking at the stack for this application we have that shown in Figure 11.2.

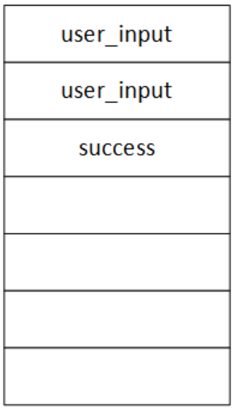


Figure 11.2: Stack for Dangerous Code Example 1

So what is the problem here? Well look at how the stack is laid out. What happens when we read into `user_input`? The danger is that is too much data is read in (via `gets` on line 17) it will overwrite the value of `success`. Note that `success` is set to 0 (false) at the start of the application, and we only change is to 1 (true) when the password is correct. The problem is, `gets` may overwrite the value of `success`. For `success` to be determined as true, it just needs to be non-zero (remember our discussion on data types). Thus, any suitable input from `gets` may result in the message `Root access granted` from appearing.

OK it is not as bad as all that. Modern C and C++ compilers recognise various problems with stack overwriting and try and protect against it. This is typically referred to as a *stack guard*. By default, compilers will have this switched on. So to break our code we have to switch it off. To do so, compile the program as follows:

```
cl filename /GS-
```

Listing 11.3: Compiling an Application with the Stack Guard Switched Off

If you run this application you can test out some basic behaviour:

```
Enter password: hello
Login failed

Enter password: password
Login successful
Root access granted

Enter password: xxxxxxxxxx
Login failed
Root access granted
```

Listing 11.4: Running Dangerous Code Example 1

Oops. Looks like we granted root access to a user who didn't successfully login. Oh well, I'm sure nothing bad will happen. . . .

Rule 1 - Don't use `gets` (or other unbounded operations)

This is our first rule when writing good quality code in C and C++. You might think that we had to manipulate things to make this work but it is a toy example. The compiler may not pick up that you have written the above vulnerability if the example is complex.

Also – for the copy-and-paste Stack Overflow crowd – how often do you blindly follow instructions and copy code? You may switch off the stack guard because of an example you found on the Internet. *If you don't know what you are doing, then you can create vulnerabilities in your code!*

Default Compiler Flags on Microsoft's Compiler

We can test the default compiler flags that `cl` uses as follows:

```
  cl filename /Bd
```

Listing 11.5: Checking Default Compiler Flags

Most of these are going into a depth that we are not covering in the module. However, there are a few ones worth noting:

W Warnings off. By default `cl` turns off warnings. Turning these back on can help write better code. You can even turn warnings into compiler errors (`WX`), although if you do so with the maximum warning level you'll find that the standard library code will stop your programs from compiling.

GS Stack guard on. This is what we talked about above.

Zp8 Data is aligned to 8 bytes. This is about how memory is laid out for optimal performance. This will become important if you study anything which requires fast code.

Ot Generate fast code (optimisation). Optimisation can be turned off, or produce small code, or both. Optimisation effectively reorders the instructions and memory to suit requirements. This can be advantageous but make debugging harder.

Ob0 Disables inline function expansion (doesn't replace functions by inserting code). This is an important optimisation that can be switched on to allow faster yet larger code to be produced.

MT Create multithreaded application (even if you don't need them). You hopefully remember threads from computer systems. By default all Windows applications have multithreaded support turned on.

We will not concern ourselves with these flags beyond the stack guard, but you should be aware of what the compiler is doing to produce your code.

11.2.3 Acting Like a Superuser

Another problem that can occur is when we let a program act as the administrator or root. When a program does this, any other process it creates also has those rights. So what does that mean? Well in C we can execute external applications using the `system` command. An example is shown in the code below.

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <string.h>
 4
 5 void get_name(char *name)
 6 {
 7   printf("Enter your name: ");
 8   gets(name);
 9 }
10
11 int main(int argc, char **argv)
12 {
13   // We are going to read in our name and run notepad.
14   char name[16];
15   char program[32];
16   strcpy(program, "notepad");
17
18   // Get the name of the user.
19   get_name(name);
20
21   // Run notepad -- DANGER! What are we actually running?
22   system(program);
23
24   return 0;
25 }
```

Listing 11.6: Dangerous Code Example 2

Note that at the start of the program we copy `notepad` into our execution string (line 16) and this command is executed as a new process (line 22). When we read in the name we can overwrite this execution string. Problems occur when we write to sections of the file system we aren't meant to. For example:

add example execution

– This one stopped working? Need to investigate.

When run as an administrator (right-click the application and run as administrator) you will create a file in the Windows folder. On a Unix-based system this can be more dangerous if the application has root privileges.

11.2.4 Stack Behaviour During Function Calls

We also discussed setting the stack when we worked with inline assembly. We did this for passing parameters into a call. Well, a bit more happens than that. For a function to return at the end of the call it needs a return address. This value is also stored on the stack, and can also be overwritten. Consider Figure 11.3.

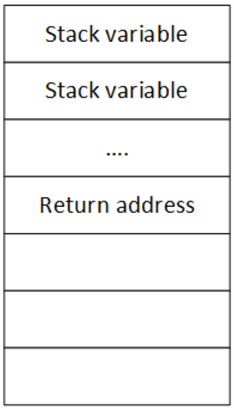


Figure 11.3: Overwriting a Return Address

If we can overwrite the return value with the address of some other piece of code then it will be executed instead. Can we do this? Yes! And all we need is the address of the code we want and Python.

11.2.5 Stack Smashing – Running Uncalled Code

The following program is another toy example where we get the addresses of our code directly. In practice, hackers will use debug tools (like we have already used) to find these values. We are just doing it in a simpler manner.

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

void copy_string(const char *input)
{
  char buf[8];

  // This prints out the current stack
  printf("My stack is:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");

  // Copy raw user input into the buffer - VERY DANGEROUS!
  strcpy(buf, input);
  printf("%s\n", buf);

  // Print the stack again
  printf("My stack is:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
}

```

```
void uncalled()
{
  printf("How was this called?!?");
}

int main(int argc, char **argv)
{
  // We could get the function addresses using a debugger but we will cheat!
  printf("Address of main = %p\n", main);
  printf("Address of copy_string = %p\n", copy_string);
  printf("Address of uncalled =  %p\n", uncalled);

  char input[8];
  gets(input);

  copy_string(input);
  return 0;
}
```

Listing 11.7: Running Uncalled Code Example

So let us run the application. The following gives an example:

```
Address of main = 00131070
Address of copy_string = 00131000
Address of uncalled =  00131050
xxxx
My stack is:
FFFFFFFF
00000001
00AFFE80
001310C1
00AFFE78
78787878
00131100
00AFFEC8

xxxx
My stack is:
78787878
00000000
00AFFE80
001310C1
00AFFE78
78787878
00131100
00AFFEC8
```

Listing 11.8: First Test of Uncalled Code Example

The key points to note is where the return address is on the stack. It is the fourth entry down (`001310C1` in this example). With that information, we can send input into the application that calls this code. We do this via Python, which allows us to send a hex string to the application by piping data. Data is pushed in reverse order on Windows, so we have to flip the input. To call `uncalled` we just use its address – `00131000`. The following is an example (replace memory address accordingly):

```
> python -c "print(’x’ * 12 + ’\x50\x10\x13\x00’)" | exploit

Address of main = 00131070
Address of copy_string = 00131000
Address of uncalled =  00131050
My stack is:
FFFFFFFF
00000001
00EFF87C
001310C1
```

```
00EFF874
78787878
78787878
78787878

xxxxxxxxxxxx
My stack is:
78787878
78787878
78787878
00131050
00EFF874
78787878
78787878
78787878

How was this called?!?
```

Listing 11.9: Exploiting Uncalled Code Example

And there you go. You have now executed code that should not have been. This could be anything, and if the program has administrator privileges then things are worse.

11.3 Some CERT Coding Standards

The point of this unit is to introduce some coding standards. We will use the CERT set of rules(`https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards`). These are defined to support secure and safe code. A number of languages are defined, including C and C++. We will be using C. You should bookmark this resource as it will be useful for you in the future.

For this section we will only look at a small number of rules. This will give an introduction to some problems that can occur. Some you may have even experienced. You should review the rules and apply them. These rules will make you a better programmer. We won't look at the solutions to these problems. To do this you should review the rules online. Solutions are available there.

11.3.1 STR30-C Do not attempt to modify string literals

A string literal is a string that you have hard-coded into your program. For example:

```c
char *str = "hello world!";
```

This string is stored specially in memory rather than as a variable. It is actually defined in the resulting assembly and machine code. This means that it is in a section of memory you cannot modify. So if you try, as the example below, you will get an error.

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

void modify(char *str)
{
  // Try to modify the string
  str[0] = 'm';
}

int main(int argc, char **argv)
{
  char *str = "hello world!";
  printf("str = %s\n", str);
  modify(str);
  return 0;
}
```

Listing 11.10: String Literal Error

The C Standard Library has a number of functions that manipulate strings. If you call them with a string literal your program will also fail. So ensure you recognise when you want to work with a string variable rather than a constant.

11.3.2 STR31-C Guarantee that storage for strings has sufficient storage for character data and the null terminator

We have already covered the problems of having strings without null-termination. If you cannot remember, review Section 1.3.

Without sufficient storage for storing string data and the null-terminator we get undefined behaviour. If we try and print a string that is not null-terminated we end up printing out extra memory. This can be dangerous depending on what is in the memory, such as password data.

11.3.3 STR32-C Do not pass a non-null-terminated character sequence to a library function that expects a string

This is a follow-on from the above rule. If you pass a non-null-terminated string to a library then it will suffer the same problem. In fact that is what we have shown with `printf` – it is a library function that expects a null-terminated string.

11.3.4 STR34-C Cast characters to `unsigned char` before converting to larger integer sizes

We have also covered the problem with casting data back in Section 2.7. This is a problem when we are working with character data and working with library functions that expect `int` data. Due to the values representing characters being signed, they will convert into incorrect data values. This can cause problems when reading a file (the `EOF` marker might be evaluated incorrectly) or when working with signals in general.

11.3.5 STR38-C Do not confuse narrow and wide character strings and functions

This is an area we have not covered: wide-character data. You are familiar with ASCII strings which are represented by characters of values from 0 to 127. But that does not cover the complete character set. If you have accented characters (such as those in many European languages) or non-Latin characters (such as Chinese, Arabic, etc.) then we need more characters. This is done by proving Unicode data, and using wide-character data. Wide-character data is a character that is 8-bits, 16-bits, or 32-bits rather than just 8-bits. It is declared as follows:

```c
wchar_t c;
```

So what is the problem here? Well, if you treat data as normal-character (narrow) rather than wide-character data in functions it will cause a problem. The following provides an example:

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

int main(int argc, char **argv)
{
  wchar_t *wstr1 = L"hello world!";
  wchar_t wstr2[13];
  printf("%s\n", wstr1);
  printf("%ls\n", wstr1);
  strncpy(wstr2, wstr1, 13);
  printf("%ls\n", wstr2);

  return 0;
}
```

Listing 11.11: Working with Wide-character Data

If you run this you will get the following output:

```
h
hello world!
h
```

Listing 11.12: Output from Wide-character Data Example

So what is happening? The first `printf` call treats the wide-character-string as a normal-string (uses `%s`). This means that it encounters what looks like a null-terminator immediately in the `h` character. Its bit pattern is as follows:

```
01101000 00000000
```

The second 8-bits are all zero, which is null.

The second `printf` is the correct way to print the string using the wide-character-string placeholder (`%ls`).

The third `printf` happens after the `strncpy` command. This also gets hit by the incorrect use of a normal-string function. The `wcsncpy` (Wide-Character String-N CoPY) function is the correct one.

11.3.6 FIO30-C Exclude user input from format strings

We will now look at another exploit – the format-string exploit. This is a big problem when working with functions such as `printf` and `fprintf`. The following code provides an example:

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

int main(int argc, char **argv)
{
  char buf[100];
  int x = 1;
9    snprintf(buf, sizeof(buf), argv[1]);
10   buf[sizeof(buf) - 1] = '\0';
11   printf("Buffer is: (%d) \nMemory address for buf: (%p) \nData input: %s \n", strlen(buf), buf, buf);
12   printf("X equals: %d/ in hex: %#x\nMemory address for x: (%p) \n", x, x, &x);
13   return 0;
14 }
```

Listing 11.13: Format-string Exploit Example

This program will copy input from `argv[1]` (remember `argv[0]` is the program name) into a buffer (line 9) and then prints it out (line 11). Let us look at some examples.

```
> format-string "Bob"
Buffer is: (3)
Memory address for buf: (0x7ffc6a29c1f0)
Data input: Bob
X equals: 1/ in hex: 0x1
Memory address for x: (0x7ffc6a29c1ec)

> format-string "Bob %p %p"
Buffer is: (33)
Memory address for buf: (0x7ffd7c5c4eb0)
Data input: Bob 0x7ffd7c5c4eb0 0x7fdb908acc60
X equals: 1/ in hex: 0x1
Memory address for x: (0x7ffd7c5c4eac)
```

Listing 11.14: "Running the Format-string Exploit

How did the second example end up printing memory addresses? Well, the `%p` in the input string was used by `printf` as a placeholder, and it printed out parts of the stack (we saw this problem earlier). We now know some useful memory addresses that we could exploit (such as the address of the buffer).

Basically, you shouldn't print out user-input using `printf` or `fprintf` (I know, we did this liberally in the workbook). When working with user-input, use `fputs` instead.

11.3.7 FIO34-C. Distinguish between characters read from a file and `EOF` or `WEOF`

The `getchar` function can be a source of problems. It returns an `int` value rather than a `char`. This is to allow certain control errors to be returned. One value it may return is a value equalling `EOF`. The `EOF` value is negative and not a character, but when cast to an `int` it might not be. As such, you should actually check you have reached the end-of-file marker by calling a function:

```
if (feof(stdin)) ...
```

This will return true only if the file end point has been reached. Otherwise, an error is likely to have occurred.

11.3.8 FIO37-C. Do not assume that `fgets()` or `fgetws()` returns a nonempty string when successful

This problem occurs when reading in data and then using `strlen` or similar. For example:

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

void func(void)
{
  char buf[1024];

  if (fgets(buf, sizeof(buf), stdin) == NULL)
  {
    /* Handle error */
  }
  buf[strlen(buf) - 1] = '\0';
}
```

Listing 11.15: Bad Use of `strlen`

The issue is that we believe the string does not start with a null-terminator. If the first character is a null-terminator (we hit return for the input) then we are setting `buf[0 - 1]` to null. This is not -1 as the size is unsigned. Hence, we are attempting to set a high-memory location to null. The program will likely crash.

11.3.9 MSC24-C. Do not use deprecated or obsolescent functions

This one is probably the most important. There are a number of Standard Library functions you should not use. We have used these in our code before, so you should be going back through and correcting our mistakes. There is only one deprecated function in C at the moment, and that is `gets`. We already covered this. The other functions are obsolescent and are listed below:

- `asctime`
- `atof`
- `atoi`
- `atol`
- `atoll`
- `ctime`
- `fopen`
- `freopen`
- `rewind`
- `setbuf`

11.4 Some Tools and Useful Resources

Thankfully, compilers and supporting tools have got better in the last few years, and the detection of errors has improved. A number of tools are available, and a number of useful websites. We list some here in no particular order:

- Open Web Application Security Project (OWASP) `https://www.owasp.org/index.php/Main_Page`.
- CERT Coding Standards `https://www.cert.org/secure-coding/`
- Clang Static Analyser `https://clang-analyzer.llvm.org/`
- Parasoft C/C++ Test `https://www.parasoft.com/product/cpptest/`

You should be working on improving your coding standards now, and on integrating these tools into your workflow. Security and secure coding are important skills with a lot of money behind them just now. Putting in our efforts here will make you a more valuable programmer in the long-run.

11.5 Exercises

As with the previous unit, there are no exercises. You should ensure that your work is also meeting these standards.