



EDINBURGH NAPIER UNIVERSITY

## **SET07109 Programming Fundamentals**

---

### **Workbook 2021/22**

---

Updated and maintained by Dr Simon Powers  
Original version created by Dr Kevin Chalmers

---

# Contents

	Page
<b>1 Compiling and Linking Programs</b>	<b>1</b>
1.1 Getting Setup	1
1.1.1 Downloading Notepad++	1
1.1.2 Running from the Command Line	2
1.2 Hello World! What Else?	3
1.2.1 The C Programming Language	4
1.2.2 Hello World Program	4
1.2.3 Building your Application	6
1.3 C Strings	8
1.4 Reading your Name	11
1.4.1 Exercise	13
1.5 Conditionals and Loops	14
1.5.1 Exercises	18
1.6 case Statements - Creating a Menu	18
1.6.1 Exercise	20
1.7 Functions	20
1.7.1 Exercise	24
1.8 Command Line Arguments	24
1.9 Compiling a Source File	25
1.10 Linking an Object File	26
1.11 Outputting Assembly Code	27
1.12 Exercises	29
<b>2 Data Sizes and Data Representation</b>	<b>31</b>
2.1 Getting the Size Of Data Using <code>sizeof</code>	31
2.2 Unsigned Data Types	32
2.3 Minimum and Maximum Values of Data Types	33
2.4 Different Datatypes	34
2.5 Compiling for 64-Bit	36
2.6 Strings	37
2.7 Casting Between Types	39
2.8 Defining <code>structs</code>	42
2.8.1 A <code>struct</code> in Memory	42
2.8.2 Accessing <code>struct</code> Members	43
2.8.3 Declaring a <code>struct</code> Variable	43
2.8.4 Passing <code>struct</code> Values as Parameters	43
2.8.5 <code>struct</code> Test Application	43
2.9 <code>enum</code> Values and <code>case</code> Statements	44
2.10 <code>const</code>	46

2.11 static Values . . . . .	47
2.12 Make Files . . . . .	48
2.12.1 Simple Make File . . . . .	48
2.12.2 Building All . . . . .	49
2.12.3 Cleaning Up . . . . .	50
2.13 Exercises . . . . .	50
<b>3 Inline Assembly</b>	<b>51</b>
3.1 First Inline Assembly Application . . . . .	51
3.2 Second Inline Assembly Application . . . . .	55
3.3 Using Assembler Operations . . . . .	56
3.3.1 Exercises . . . . .	57
3.4 Using the Stack . . . . .	58
3.4.1 What is the Stack? . . . . .	58
3.4.2 Working with the Stack . . . . .	59
3.4.3 Test Application . . . . .	59
3.5 Calling Functions - printf . . . . .	60
3.5.1 Calling Procedures in Assembly . . . . .	61
3.5.2 Setting the Stack . . . . .	61
3.5.3 Clearing the Stack . . . . .	61
3.5.4 Example Application . . . . .	61
3.5.5 Exercise . . . . .	63
3.6 Calling Functions - Writing your Own . . . . .	63
3.7 For the Brave - Loops . . . . .	66
3.7.1 Exercise . . . . .	69
3.8 Why did we Just Look at Assembly? . . . . .	69
3.9 Additional Resources . . . . .	69
3.10 Exercises . . . . .	70
<b>4 Including Files and Declaration Order</b>	<b>71</b>
4.1 The Pre-Processor . . . . .	71
4.1.1 Some Pre-Processor Commands . . . . .	71
4.1.2 Defining Values at Compile Time . . . . .	73
4.1.3 Using #ifdef for Conditional Compilation . . . . .	73
4.1.4 Exercise . . . . .	74
4.2 Creating a Header File . . . . .	74
4.3 Compiling Multiple Files into One . . . . .	76
4.3.1 Compiling Multiple Files . . . . .	76
4.3.2 Example - Student Details . . . . .	77
4.3.3 Exercises . . . . .	78
4.4 Creating and Linking Libraries . . . . .	79
4.4.1 Compiling Code into a Library . . . . .	79
4.4.2 Linking to a Library . . . . .	79
4.4.3 Exercise - Student Details (Again) . . . . .	80
4.5 A Simple Array Library . . . . .	80
4.5.1 search.h . . . . .	80
4.5.2 search.c . . . . .	81
4.5.3 sort.h . . . . .	81
4.5.4 sort.c . . . . .	82
4.5.5 generate.h . . . . .	83

4.5.6	<code>generate.c</code>	83
4.5.7	Test Application	84
4.5.8	Compiling the Array Library	85
4.5.9	Output from Array Library Test Application	86
4.6	Reading Files	86
4.6.1	Opening a File	86
4.6.2	Reading Text Files	87
4.6.3	Tokenising (splitting) strings	89
4.6.4	Reading a Binary File	90
4.6.5	Reading a Binary File	91
4.6.6	Exercise	93
4.7	Writing Files	93
4.8	Exercises	94
<b>5</b>	<b>Call Conventions - Passing by Value, Reference, and Pointer</b>	<b>95</b>
5.1	The C++ Programming Language	95
5.2	Hello World in C++	96
5.3	Working with <code>std::string</code> in C++	96
5.3.1	Exercise	98
5.4	Accessing Raw String from C++ <code>string</code>	98
5.5	Reading Input from the Command Line with <code>cin</code>	99
5.6	Using <code>getline</code> to Read Lines of Text	100
5.7	Passing by Value (Copying Data)	101
5.8	Passing by Reference	102
5.9	Example - Copying and Sorting by Value and Reference	105
5.9.1	Exercise	107
5.10	Array Library in C++	107
5.10.1	Sorting	108
5.10.2	Generating	109
5.10.3	Test Application	110
5.10.4	Exercise	110
5.11	Reading Files in C++	111
5.12	Writing Files in C++	113
5.13	<code>const</code> References	114
5.13.1	Exercise	115
5.14	Pointers	115
5.15	Arrays as Pointers to Memory	117
5.16	<code>const</code> Pointers and Pointers to <code>const</code>	119
5.17	Namespaces	121
5.18	Using C Libraries in C++	123
5.19	Exercises	124
<b>6</b>	<b>Memory Management - Using the Stack and Heap</b>	<b>125</b>
6.1	Scope of Values	125
6.1.1	Which <code>x</code> is in Scope?	126
6.1.2	Values out of Scope	127
6.1.3	Losing Values on the Stack	128
6.2	Allocating Data in Global Memory (the Heap)	130
6.2.1	The <code>malloc</code> Function	131
6.2.2	The <code>free</code> Function	131

6.2.3	Using <code>calloc</code> to Clear Memory While Allocating	132
6.2.4	Example	132
6.3	Limits of the Stack	134
6.3.1	<b>For the Brave</b> - Setting the Stack Size with the Linker	135
6.4	Allocating Large Memory Blocks on the Heap	135
6.4.1	Exercise	136
6.5	Using Pointers to Pointers to Allocate Memory to Parameters	136
6.6	Using <code>new</code> in C++ to Allocate Memory	138
6.6.1	Using <code>new</code> to Allocate Memory	139
6.6.2	Using <code>delete</code> to Free Memory	139
6.6.3	Using <code>new</code> to Allocate Arrays	140
6.6.4	Freeing Arrays with <code>delete[]</code>	140
6.6.5	Test Application	140
6.7	Best Practice When Working with Allocated Memory	142
6.8	<code>shared_ptr</code> and Automatic Memory Management	142
6.9	Examining <code>shared_ptr</code>	145
6.10	Using <code>shared_ptr</code> for Arrays	147
6.11	<code>unique_ptr</code>	148
6.12	<code>unique_ptr</code> for Arrays	150
6.13	The <code>auto</code> Keyword	151
6.14	<b>For the Brave</b> - Casting in C++ Using <code>static_cast</code>	152
6.15	<b>For the Brave</b> - Casting with <code>const_cast</code>	153
6.16	<b>For the Brave</b> - Using <code>typeid</code>	154
6.17	Case Study - Building Trees	155
6.17.1	Trees	155
6.17.2	Building a Binary Tree	156
6.17.3	Getting Started with Binary Trees	157
6.17.4	Inserting a Value into a Empty Tree	159
6.17.5	Challenge 1 - Printing In Order	160
6.17.6	Challenge 2 - Inserting into a Non-Empty Tree	160
6.17.7	Challenge 3 - Freeing Resources	161
6.17.8	Exercises	161
<b>7</b>	<b>Object-Orientation and C++</b>	<b>163</b>
7.1	What is Object-Orientation?	163
7.2	Using <code>struct</code> in C++	164
7.3	Data Sizes and Memory Representation of <code>struct</code>	165
7.3.1	Exercise	167
7.4	Defining a <code>class</code>	168
7.5	Data Size of a <code>class</code>	169
7.5.1	Exercise	170
7.6	<code>public</code> and <code>private</code>	171
7.7	Defining Methods	173
7.8	Object Construction	176
7.9	Object Construction Order	178
7.9.1	Exercise 1	182
7.9.2	<b>For the Brave</b> - Exercise 2	183
7.10	Object Destruction	183
7.10.1	Exercise	187
7.10.2	Scope Revisited	187

7.11 Creating Objects on the Heap	187
7.11.1 Exercise	190
7.12 Accessing Members of Pointers	190
7.12.1 Dereferencing the Pointer	190
7.12.2 Using the -> Operator	192
7.13 Inheritance	193
7.13.1 Exercise	197
7.14 protected Class Members	198
7.14.1 Exercise	200
7.15 For the Brave - static class Members	200
7.16 For the Brave - Casting with dynamic_cast and reinterpret_cast	202
7.17 For the Brave - Writing and Reading struct and class Data	204
7.18 Exercises	206
<b>8 Virtual Function Calls</b>	<b>207</b>
8.1 Overridable Behaviour in Classes	207
8.2 The virtual Keyword in C++	209
8.3 virtual Methods	209
8.4 virtual Method Tables	210
8.5 Inheritance and Destruction	211
8.6 virtual Destructors	213
8.7 Overriding Methods	214
8.7.1 override Keyword	214
8.7.2 final Keyword	219
8.8 Pure virtual Methods	220
8.8.1 Exercise	222
8.9 Pure virtual Classes - Interfaces	222
8.10 Exercises	224
<b>9 Operator Overloading</b>	<b>225</b>
9.1 Operator Overloading	225
9.2 Operator Overloading in Classes	227
9.3 A vec2 Class	229
9.4 Comparison Operators	230
9.4.1 ==	230
9.4.2 !=	231
9.4.3 <	233
9.4.4 >	235
9.4.5 <=	236
9.4.6 >=	238
9.5 Arithmetic Operators	240
9.5.1 Change in Approach	240
9.5.2 +	240
9.5.3 -	242
9.5.4 *	244
9.5.5 /	245
9.6 Assignment Operators	247
9.6.1 =	247
9.6.2 +=	249
9.7 Complete vec2 Class	251

9.8 For the Brave - Copy Constructors . . . . .	253
9.9 For the Brave - Conversion Operators . . . . .	256
9.10 For the Brave - Member Access . . . . .	257
9.11 For the Brave - Overriding the Input and Output Operators . . . . .	259
9.12 Exercise . . . . .	261
<b>10 Coding Standards for Quality and Security</b>	<b>263</b>
10.1 Why Bad Code is Bad . . . . .	264
10.2 Some Dangerous Code Examples . . . . .	264
10.2.1 Stack Behaviour . . . . .	265
10.2.2 Overwriting Data . . . . .	265
10.2.3 Acting Like a Superuser . . . . .	268
10.2.4 Stack Behaviour During Function Calls . . . . .	269
10.2.5 Stack Smashing – Running Uncalled Code . . . . .	269
10.3 Some CERT Coding Standards . . . . .	271
10.3.1 STR30-C Do not attempt to modify string literals . . . . .	271
10.3.2 STR31-C Guarantee that storage for strings has sufficient stor- age for character data and the null terminator . . . . .	272
10.3.3 STR32-C Do not pass a non-null-terminated character se- quence to a library function that expects a string . . . . .	272
10.3.4 STR34-C Cast characters to <code>unsigned char</code> before converting to larger integer sizes . . . . .	272
10.3.5 STR38-C Do not confuse narrow and wide character strings and functions . . . . .	272
10.3.6 FIO30-C Exclude user input from format strings . . . . .	273
10.3.7 FIO34-C. Distinguish between characters read from a file and EOF or WEOF . . . . .	274
10.3.8 FIO37-C. Do not assume that <code>fgets()</code> or <code>fgetws()</code> returns a nonempty string when successful . . . . .	274
10.3.9 MSC24-C. Do not use deprecated or obsolescent functions . . . . .	275
10.4 Some Tools and Useful Resources . . . . .	275
10.5 Exercises . . . . .	276
<b>11 Debugging using an Intergrated Development Environment (not assessed this year)</b>	<b>277</b>
11.1 Getting Started with Visual Studio . . . . .	277
11.1.1 Where are my Files? . . . . .	280
11.2 Starting without Debug . . . . .	280
11.3 What is a Project? . . . . .	281
11.4 Working with Multiple Project Solutions . . . . .	282
11.5 Debug versus Release Builds . . . . .	282
11.6 Setting Breakpoints . . . . .	284
11.7 Stepping Through a Program . . . . .	285
11.8 <code>assert</code> . . . . .	286
11.9 Watching Variables . . . . .	288
11.10 Advanced Breakpoints . . . . .	290
11.11 Examining the Call Stack . . . . .	291
11.12 For the Brave - Using the Disassembly . . . . .	292
11.13 For the Brave - Examining the Registers . . . . .	292
11.14 For the Brave - Examining Memory . . . . .	293



11.15 <b>For the Brave</b> - Detecting Memory Leaks . . . . .	293
11.16Exercises . . . . .	294



# List of Figures

1.1	Notepad++ Start Screen	2
1.2	Visual Studio Developer Command Prompt, which we will be using to compile and run our applications.	3
1.3	C Strings and null Termination	9
1.4	Command Line Arguments Structure	24
2.1	student struct Represented in Memory	43
3.1	Values Added and Removed from the Stack During Operation	59
3.2	Using the Stack to Swap Values with push and pop	60
4.1	Inclusion Structure	77
4.2	Structure of Student Application - both File Inclusion and Object Files	78
4.3	Library Inclusion Structure	80
4.4	Linking a Library	80
4.5	Array Library File Structure	85
5.1	Interpretation of the Stack During Pass-by-Value	102
5.2	Interpretation of the Stack Using Pass-by-Reference	104
5.3	Task Manager During Pass-by-Copy (left) and Pass-by-Reference (right)	108
5.4	Data and Stack Corruption Example - Returning Non-Valid Memory	120
6.1	Stack and Heap	131
6.2	Binary Tree Example	155
6.3	Binary Tree Start	156
6.4	Building the Left Branch	156
6.5	Data Structure of a Binary Tree	157
7.1	Class Diagram for Data Sizes Application	166
7.2	Class Diagram Showing Inheritance	193
8.1	Virtual Method Table	211
8.2	Great Grandchild Application Diagram	216
8.3	Animal Class Diagram	222
10.1	Example Stack	265
10.2	Stack for Dangerous Code Example 1	266
10.3	Overwriting a Return Address	269
11.1	Visual Studio 2019 Start Screen	278
11.2	New Project Window in Visual Studio	278
11.3	New Project Setup Configuration	279
11.4	Visual Studio with Empty Project Created	279

11.5 Debugging a Visual Studio Project . . . . .	279
11.6 Opening Project File Location in Visual Studio . . . . .	280
11.7 Compiling a File in Visual Studio . . . . .	281
11.8 Building a Project in Visual Studio . . . . .	281
11.9 Running a Visual Studio Project without Debugging . . . . .	281
11.10 Visual Studio Project Highlighted in Solution Explorer . . . . .	282
11.11 Visual Studio Project Properties . . . . .	282
11.12 Adding a Project to an Existing Solution in Visual Studio . . . . .	283
11.13 Setting the Startup Project in Visual Studio . . . . .	283
11.14 Changing Between Debug and Release Builds . . . . .	283
11.15 Setting a Breakpoint in Visual Studio . . . . .	284
11.16 Stepping into a Program in Visual Studio . . . . .	285
11.17 Debug Stepping Tools in Visual Studio . . . . .	286
11.18 Assertion Failure in Visual Studio . . . . .	287
11.19 Watching a Variable in Visual Studio . . . . .	289
11.20 Visual Studio Watch Window . . . . .	290
11.21 Setting an Advanced Breakpoint . . . . .	291
11.22 Visual Studio Breakpoint Condition Window . . . . .	291
11.23 Visual Studio Call Stack . . . . .	292
11.24 Visual Studio Disassembly . . . . .	292
11.25 Visual Studio Registers View . . . . .	293
11.26 Visual Studio Memory View . . . . .	293

# List of Algorithms

1	Prompting for and Displaying the Username	12
2	Star Printing Algorithm	16
3	Menu System	19
4	Wage Calculator	22
5	Printing Your Name Surrounded by Stars	30
6	A Loop Application to Display Command Arguments Written in As-	
	sembly	68
7	Linear Search Algorithm	81
8	Bubble Sort Algorithm	82



# 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 how programming languages work, looking at issues such as calling conventions, memory management, and pointers. The goal of the module is to enhance your understanding of these fundamental concepts that are at the core of all languages, to help you become a better programmer and be able to easily learn new languages in the future. As such, you should work through all the examples and exercises that you can. With perseverance everyone should be able to work through all the module content, regardless of how long you have been programming for.

### 1.1 Getting Setup

In this module, we will work using a text editor and compiling from the command line. The toolchain we are going to use is Microsoft's set of C/C++ compilers, linkers and assemblers. However, you can use other C/C++ compilers such as `gcc` and `clang` if you are a Linux or Mac OS X user. We will give examples of using `clang` in the **On a Mac?** sections (the commands for `gcc` on Linux are largely the same – ask on Teams if you get stuck). The code we are using in this module is standard C/C++, and we avoid using Microsoft specific libraries. The exception is for Unit 3, where we look at inline assembly language in order to get a better understanding of how C code translates into assembly language, and to look at what happens at the machine level when you call a function in your code. For Unit 3 only, we recommend using the Microsoft compiler even if you are on Mac or Linux. This can be accessed through Apps Anywhere on the University's virtual desktop (<https://desktop.napier.ac.uk/>).

To install Visual Studio on your own machine, follow the instructions on Moodle under the “Activities for Week 1”. On a Mac, follow the instructions for installing `clang`, also on Moodle in the “Activities for Week 1” section. Visual Studio 2019 is already installed on university machines and is accessed through *Apps Anywhere* as described below.

#### 1.1.1 Downloading Notepad++

If you are using Microsoft Windows, the text editor we recommend for the module is Notepad++ (<https://notepad-plus-plus.org/>). Notepad++ is free, provides syntax 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*. On a Mac you

can use Atom (<https://notepad-plus-plus.org/>). However, you are free to use any text editor that you like on this module. One easy to use alternative on both Windows and Mac is Visual Studio Code (<https://code.visualstudio.com/>) – if you use this, remember to add the C++ extension.

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

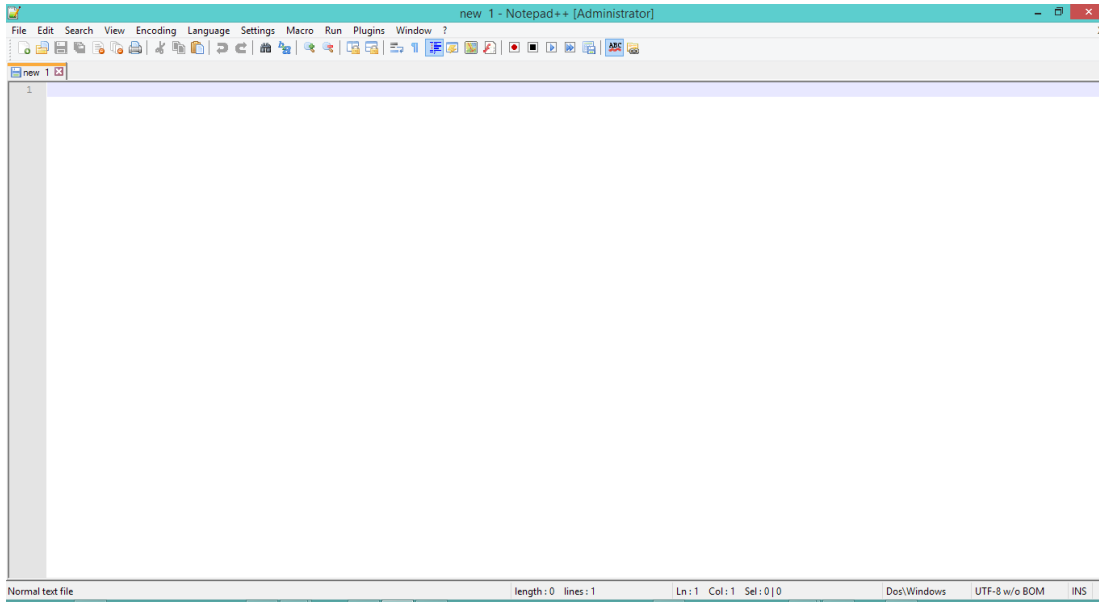


Figure 1.1: Notepad++ Start Screen

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 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. On university machines you should use your H drive. On your home machine your University Microsoft OneDrive is a good option, so that your work is backed up (especially your coursework!). Otherwise, you could store your work locally in your documents folder (but make sure to back it up somewhere!).

### 1.1.2 Running from the Command Line

We will be working from the command line to compile and run our applications, in order to gain a deeper understanding of the build process (i.e. what goes on behind the scenes when you press the big green button in an Integrated Development Environment such as Eclipse or Visual Studio). This means you need to open the Visual Studio Developer Command Prompt.

To do this on University machines, you first need to start Visual Studio 2019 through *Apps Anywhere*. When you get to the sign-in screen you can ignore it, but then go to the Visual Studio 2019 folder on the start menu, and choose Developer Command Prompt for VS 2019 (usually the second item from the top).

**On a Mac?** Start a terminal window (this can be done by pressing cmd space to bring up Spotlight, then type terminal).

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



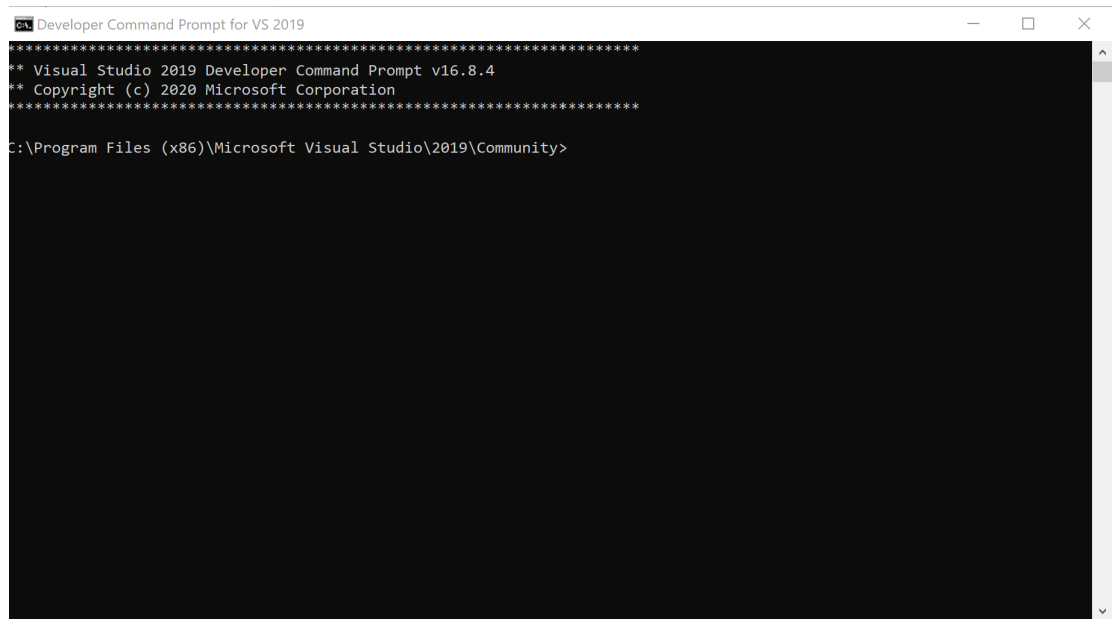


Figure 1.2: Visual Studio Developer Command Prompt, which we will be using to compile and run our applications.

your H drive on a university machine, under a subfolder called **SET07109**. The commands to use are (press enter after each command to execute it):

```
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**. The **cd** command is used to *change directory* (directory is a more technical name for folder). If you need to go up a directory in the folder structure, you can use the **cd ..** command. If you want to get a list of all of the files in the current directory, then the command is **dir**. You can find out more about using the command prompt at <http://dosprompt.info/>

**On a Mac?** When you open a terminal window you will start in your home directory. The command to change directory (e.g. into your Documents directory) is also **cd**. The command to get a list of all of the files in the current directory is **ls**.

**Hint** On both Mac and Windows, you can also first find the folder that you would like to move in with the graphical user interface (Explorer on Windows and Finder on Mac). You can then type **cd** followed by a space into the command prompt, and then drag the folder into the command prompt. This will fill in the path to the folder for you.

## 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.

### 1.2.1 The C Programming Language

C is one of the most popular programming languages in the world (as of January 2022 it is the second most popular: <https://www.tiobe.com/tiobe-index/>). 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 and C#), 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.

Because C captures the concepts common to most languages, we will be using it (and later C++) in this module. The understanding that you develop of C will translate into a better understanding of how more modern “managed” languages work. For example, our detailed look at memory management in C and C++ will give you a deeper understanding of what is going on behind the scenes in Java and C#, and how what these languages are doing behind the scenes can affect the performance of the applications that you write. This will help you in the future to both code more effectively, and to make an informed decision about the most suitable language to use. It will also give you an understanding of the core principles of programming languages that will allow you to easily learn new languages that are developed throughout your career.

### 1.2.2 Hello World Program

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 entered into your text editor (e.g. Notepad++) saved in a file called `hello.c`. The code is given below. Do not try and copy the code from the PDF – this can introduce errors, and you will learn far more by entering the code yourself and thinking about what you are writing.

```
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 1.1: 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. .

#### `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 Brian and `age` is 79 then it prints 'Hello Brian. You are 79 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](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, then it's fine to use that. The key is to be consistent throughout all of the code that you write.

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.

Finally, your commenting style matters. The examples in this Workbook include comments on most lines. This is only for teaching purposes, to explain to you what they do. You should never normally comment every line, as most lines will be self explanatory for programmers familiar in C. In this module, you should adopt the following commenting standard:

1. Include a comment at the top of the file giving the author name and the purpose of the code in the file.
2. Include a comment above every function describing the purpose of the function, what it returns, and the meaning of the parameters that it accepts.
3. Comment any complex section of code, e.g. it's often useful to put a comment above a for loop to explain what that loop will be doing.

Variable names should always begin with lower case letters.

### 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 19.28.29336 for x86
Copyright (C) Microsoft Corporation. All rights reserved.

hello.c
Microsoft (R) Incremental Linker Version 14.28.29336.0
Copyright (C) Microsoft Corporation. All rights reserved.

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

Listing 1.3: Output from Building our Hello World Application

**On a Mac?** The command to compile your Hello World application is this:

```
clang -o hello hello.c
```

Listing 1.4: Building our Hello World Application using clang on a Mac

The `-o` argument specifies the output file name (“hello” in this case).

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 1.5: 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 1.6: 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 correctly, and then run it by typing `hello` at the command prompt:

```
hello
Hello world!
```

Listing 1.7: Running the Hello World Application

**On a Mac?** Run your application by typing `./hello`.

## 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.

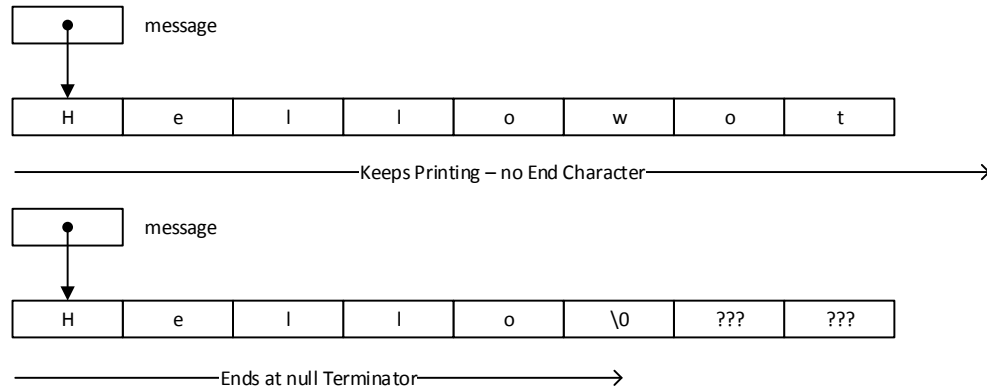


Figure 1.3: C Strings and null Termination

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.

```

1 #include <stdio.h>
2 // contains helper functions for strings
3 #include <string.h>
4
5 int main(int argc, char **argv)
6 {
7     // Declare a character array for message
8     char msg_1[5] = {'H','e','l','l','o'};
9     // Declare a second character array for message
10    char msg_2[8] = " World!";
11    // Declare a third message
12    char msg_3[9] = "Goodbye!\0";
13    // Declare a forth message - no size
14    char msg_4[] = "Compiler worked out my size!";
15    // Declare a fifth message - use a pointer
16    char *msg_5 = "Compiler worked out my size too!";
17
18    // Print messages. 1st message is not null-terminated
19    printf("%s\n", msg_1);
20    printf("%s\n", msg_2);
21    printf("%s\n", msg_3);
22    printf("%s\n", msg_4);
23    printf("%s\n", msg_5);
24
25    return 0;
26 }
```

## Listing 1.8: 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:

```
1 int x = 42;
2 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 1.9: 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.

---

**Algorithm 1** Prompting for and Displaying the Username

---

```
1: procedure MAIN
2:   prompt for first name
3:   read first name
4:   add null terminator to first name
5:   if first name == 'Brian' then
6:     print hello Brian message
7:   else
8:     print disappointed message
9:   prompt for last name
10:  read last name
11:  add null terminator to last name
12:  add first name to full name
13:  add space to full name
14:  add last name to full name
15:  print full name
16:  print length of full name
```

---

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

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(int argc, char **argv)
5 {
6     // Declare character arrays to store name
7     char first_name[50];
8     char last_name[50];
9     char full_name[50];
10
11     // Prompt for first name
12     printf("Please enter your first name: ");
13     // Read name into array from stdin (standard input)
14     fgets(first_name, 50, stdin);
15
16     // Remove the newline character
17     // Gets the length of the string. Finds the null terminator
18     int len = strlen(first_name);
19     // Check if last character is a newline
20     if (len > 0 && first_name[len - 1] == '\n')
21         // If so set newline character to null terminator
22         first_name[len - 1] = '\0';
23
24     // Check if your name is Brian
25     if (strcmp(first_name, "Brian") == 0)
26         printf("Hey! Another Brian\n");
27     else
28         printf("Oh well\n");
29
30     // ***** DO THE SAME FOR LAST NAME *****
31
```

```

32     // Join the strings
33     // strcat looks for null terminator. Set first character of
        full_name
34     full_name[0] = '\0';
35     // Add first name to full name
36     strcat(full_name, first_name);
37     // Add a space
38     strcat(full_name, " ");
39     // Add last name
40     strcat(full_name, last_name);
41     // Print name
42     printf("Your full name is %s which is %d characters long\n",
        full_name, strlen(full_name));
43
44     return 0;
45 }

```

Listing 1.10: 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: Brian
Hey! Another Brian
You entered 5 characters
Please enter your last name: Kernighan
You entered 9 characters
Your full name is Brian Kernighan which is 15 characters long

```

Listing 1.11: 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.

```
1 if (condition)
2 {
3     // If condition is true this code is run
4     ...
5 }
```

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:

```
1 if (condition)
2 {
3     // If condition is true this code is run
4     ...
5 }
6 else
7 {
8     // If condition is false this code is run
9     ...
10 }
```

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

```

1  if (condition1)
2  {
3      // If condition1 is true this code is run
4      ...
5  }
6  else if (condition2)
7  {
8      // If condition1 is false and condition2 is true this code
9      // is run
10     ...
11 }
12 else
13 {
14     // If condition1 is false and condition2 is false this code
15     // is run
16     ...
17 }

```

`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

---

```
1: procedure MAIN
2:   flag  $\leftarrow$  1
3:   while flag = 1 do
4:     prompt for number of stars (number)
5:     read in number of stars (number)
6:     if number = 0 then
7:       flag  $\leftarrow$  0
8:       continue
9:     for i  $\leftarrow$  0 to number do
10:      for j  $\leftarrow$  0 to i do
11:        print *
12:      print new line
13:   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:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(int argc, char **argv)
6 {
7     // Flag (1 or 0) to indicate if we should continue
8     int flag = 1;
9     // while loop
10    while (flag)
11    {
12        // Character array to read number into
13        char buffer[10];
14        // Prompt for value
15        printf("How many stars? (0 to quit) ");
16        // Read in value
17        fgets(buffer, 10, stdin);
18        // Convert string to number
19        int number = atoi(buffer);
20        // If number equals 0 then set flag and continue
21        if (!number)
22        {
23            flag = 0;
24            continue;
25        }
26        // Print stars
27        for (int i = 0; i < number; ++i)
28        {
29            for (int j = 0; j < i; ++j)
30                printf("*");
31            printf("\n");
32        }
33    }
34    // Say goodbye
35    printf("Goodbye!\n");
36
37    return 0;
```

## Listing 1.12: 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
6     // in the if block
7     // will run!
8 }
```

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 its 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 1.13: 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

---

```

1: procedure MAIN
2:    $flag \leftarrow 1$ 
3:   while  $flag = 1$  do
4:     print menu
5:     read input ( $number$ )
6:     switch  $number$  do
7:       case 1
8:         print hello
9:       case 2
10:        print goodbye
11:      case 0
12:        print exit
13:         $flag \leftarrow 0$ 
14:      case default
15:        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:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(int argc, char **argv)
6 {
7     int flag = 1;
8     while (flag)
9     {
10         // Print menu
11         printf("1 - say hello\n");
12         printf("2 - say goodbye\n");
13         printf("0 - exit\n");
14         printf("Enter choice - ");
15         // Read input
16         char buffer[10];
17         fgets(buffer, 10, stdin);
18         // Convert to number
19         int number = atoi(buffer);
20
21         // Work on input
22         switch (number)
23         {
24             case 1:
25                 printf("Hello World!\n");
26                 break;
27             case 2:
28                 printf("Goodbye World!\n");
29                 break;

```

```
30         case 0:
31             printf("Exiting...\n");
32             flag = 0;
33             break;
34         default:
35             printf("*** INVALID INPUT ***\n");
36             break;
37     }
38 }
39
40 return 0;
41 }
```

Listing 1.14: 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 1.15: 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

```
1 #include <stdio.h>
2 // The main C standard library header
3 #include <stdlib.h>
4 #include <string.h>
5
6 // Calculate wage
7 double calculate_wage(double rate, double hours)
```

**Algorithm 4** Wage Calculator

---

```
1: function CALCULATE_WAGE(rate, hours)
2:   return rate × hours

3: procedure MAIN
4:   flag ← 1
5:   while flag = 1 do
6:     read name (name)
7:     if name = "0" then
8:       flag ← 0
9:       continue
10:    read rate (rate)
11:    read hours (hours)
12:    wage ← CALCULATE_WAGE(rate, hours)
13:    print wage
```

---

```
8 {
9   return rate * hours;
10 }
11
12 // Remove newline. Pass in modifiable value
13 void remove_newline(char *str)
14 {
15   // Gets the length of the string
16   int len = strlen(str);
17   // Check if last character is a newline
18   if (len > 0 && str[len - 1] == '\n')
19     // If so set newline character to null terminator
20     str[len - 1] = '\0';
21 }
22
23 int main(int argc, char **argv)
24 {
25   // Flag to continue
26   int flag = 1;
27   while (flag != 0)
28   {
29     // Buffer for name
30     char name[50];
31     // Buffer for number
32     char number[10];
33     // Prompt for first name
34     printf("Please enter your name: ");
35     // Read name into array
36     fgets(name, 50, stdin);
37     // Strip newline
38     remove_newline(name);
39     // If name is 0 then exit
40     if (strcmp(name, "0") == 0)
41     {
42       flag = 0;
43       continue;
44     }
45     // Prompt for hourly rate
46     printf("Enter hourly rate: ");
47     // Read value into array
```

```

48     fgets(number, 10, stdin);
49     // Convert to float
50     double rate = atof(number);
51     // Prompt for hours
52     printf("Enter hours: ");
53     // Read value into array
54     fgets(number, 10, stdin);
55     // Convert to float
56     double hours = atof(number);
57     // Calculate wage
58     double wage = calculate_wage(rate, hours);
59     // Output
60     printf("Wage for %s: %.2f\n", name, wage);
61 }
62 // Print goodbye
63 printf("Goodbye!");
64
65 return 0;
66 }

```

Listing 1.16: 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. We will revisit this issue in more detail in Unit 4.

An example output from this application is shown below.

```

Please enter your name: Neil
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 1.17: 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

```

cl functions.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.28.29336 for x86
Copyright (C) Microsoft Corporation. All rights reserved.

functions.c
functions.c(64) : error C2371: 'calculate_wage' : redefinition; different basic
types

```

## 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.

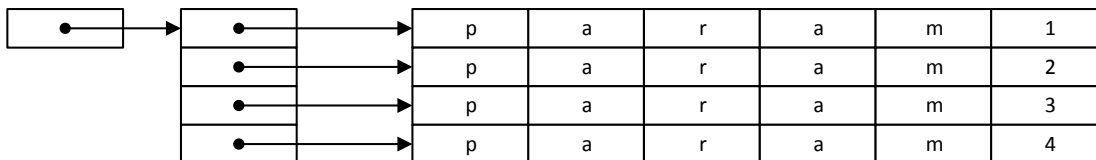


Figure 1.4: Command Line Arguments Structure

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

```

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

```

```

5 {
6     // Loop for number of arguments
7     for (int i = 0; i < argc; ++i)
8     {
9         printf("Argument %d: %s\n", i, argv[i]);
10    }
11    printf("All arguments printed.\n");
12
13    return 0;
14 }

```

Listing 1.18: 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 1.19: 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 a 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 (e.g. the library containing `printf`) 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 than 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  $\Rightarrow$  object file  $\Rightarrow$  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 1.20: Compiling to an object file

This will provide the following output:

```
cl /c hello.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.28.29336 for x86
Copyright (C) Microsoft Corporation. All rights reserved.
hello.c
```

Listing 1.21: 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.

**On a Mac?** The command to compile only and produce an object file, rather than an executable, is:

```
clang -o hello.obj -c hello.c
```

Listing 1.22: Compiling to an object file using clang on a Mac.

Here we are using the `-c` option of clang to compile to object code only, and not link in to operating system libraries to produce an executable (this is the same as the `/c` option on the Microsoft compiler).

## 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 1.23: Linking an Object File

The output from this command is shown below:

```
link hello.obj
Microsoft (R) Incremental Linker Version 14.28.29336.0
Copyright (C) Microsoft Corporation. All rights reserved.
```

Listing 1.24: Output from the 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.

**On a Mac?** You can link your produce file to produce an executable by running clang on the object file:



```
clang -o hello hello.obj
```

Listing 1.25: Linking an Object File

## 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  $\Rightarrow$  Assembly code  $\Rightarrow$  Object code  $\Rightarrow$  executable

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

```
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 }
```

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

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

Listing 1.26: Generating Assembly Code

The output from this step is as follows:

```
cl /Fa hello.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.28.29336 for x86
Copyright (C) Microsoft Corporation. All rights reserved.

hello.c
Microsoft (R) Incremental Linker Version 14.28.29336.0
Copyright (C) Microsoft Corporation. All rights reserved.

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

Listing 1.27: 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:

```
1 ; Listing generated by Microsoft (R) Optimizing Compiler Version 19
  .28.29336
2
3 TITLE D:\programming-fundamentals\code\unit-01\hello.c
4 .686P
5 .XMM
6 include listing.inc
7 .model flat
8
9 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 1.28: 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.

**On a Mac?** You can get clang to generate the assembly language using the `-S` `-masm=intel` options, e.g.

```
clang -S -masm=intel hello.c
```

Listing 1.29: Generating Assembly Code

This will output the assembly code produced by the compiler into the file `hello.s`. Note that the assembly code will look different from the listing above for the Microsoft Compiler. One reason for this is that clang is compiling to 64 bit, compared to the 32 bit x86 architecture that we are compiling for on the Microsoft Compiler (we will look at compiling 64 bit applications on the Microsoft compiler in the next unit).

## 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:

```
*****
* Brian Kernighan *
*****
```

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. Translate your Christmas Tree algorithm from the Week 1 activities into C. As a reminder, the task is to 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

---

```
1: procedure MAIN
2:   read name (name)
3:   length  $\leftarrow$  STRLEN(name) + 4
4:   for i  $\leftarrow$  0 to length do
5:     print *
6:   print new line
7:   print *
8:   print space
9:   print name
10:  print space
11:  print *
12:  print new line
13:  for i  $\leftarrow$  0 to length do
14:    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` operator to get the size of a data type in bytes.

#### The sizeof Operator

`sizeof` is a very useful operator 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.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char **argv)
5 {
6     printf("The size of int is %d bytes\n", sizeof(int));
7
8     return 0;
```

```
9 }
```

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. Note that you will receive compiler *warnings*, as we are doing things that you should not normally do to show you what happens. Because these are warnings, rather than errors, you will see that the compiling, assesmbling and linking process still complete and the executable code is generated.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char **argv)
5 {
6     int a = -500;
7     unsigned int b = -500;
8     unsigned int c = (unsigned int)a;
9     float f = -500;
10
11     printf("a signed: %d, a unsigned: %u\n", a, a);
12     printf("b signed: %d, b unsigned: %u\n", b, b);
13     printf("c signed: %d, c unsigned: %u\n", c, c);
```

```

14     printf("f signed: %d, f unsigned: %u\n", f, f);
15     printf("a == b: %d\n", a == b);
16     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:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <limits.h>
4
5 int main(int argc, char **argv)
6 {
7     printf("unsigned int is %d bytes, min value %u, max value %u\n",
8           , sizeof(unsigned int), 0, UINT_MAX);
9     printf("int is %d bytes, min value %d, max value %d\n", sizeof(
10           int), INT_MIN, INT_MAX);
11
12     return 0;
13 }
```

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.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <limits.h>
4 #include <float.h>
5
6 int main(int argc, char **argv)
7 {
8     printf("unsigned char is %d bytes, min value %u, max value %u\n",
9           , sizeof(unsigned char), 0, UCHAR_MAX);
10    printf("signed char is %d bytes, min value %d, max value %d\n",
11          , sizeof(signed char), SCHAR_MIN, SCHAR_MAX);
12    printf("char is %d bytes, min value %d, max value %d\n", sizeof(
13          (char), CHAR_MIN, CHAR_MAX);
14 }
```



```

11     printf("unsigned short is %d bytes, min value %u, max value %d\\
12         n", sizeof(unsigned short), 0, USHRT_MAX);
13     printf("short is %d bytes, min value %d, max value %d\\n",
14         sizeof(short), SHRT_MIN, SHRT_MAX);
15     printf("unsigned int is %d bytes, min value %u, max value %u\\n"
16         , sizeof(unsigned int), 0, UINT_MAX);
17     printf("int is %d bytes, min value %d, max value %d\\n", sizeof(
18         int), INT_MIN, INT_MAX);
19     printf("unsigned long is %d bytes, min value %lu, max value %lu
20         \\n", sizeof(unsigned long), 0, ULONG_MAX);
21     printf("long is %d bytes, min value %ld, max value %ld\\n",
22         sizeof(long), LONG_MIN, LONG_MAX);
23     printf("unsigned long long is %d bytes, min value %llu, max
24         value %llu\\n", sizeof(unsigned long long), 0, ULLONG_MAX)
25         ;
26     printf("long long is %d bytes, min value %lld, max value %lld\\n
27         ", sizeof(long long), LLONG_MIN, LLONG_MAX);
28     printf("float is %d bytes, min value %e, max value %e\\n",
29         sizeof(float), FLT_MIN, FLT_MAX);
30     printf("double is %d bytes, min value %e, max value %e\\n",
31         sizeof(double), DBL_MIN, DBL_MAX);
32     printf("long double is %d bytes, min value %e, max value %e\\n",
33         sizeof(long double), LDBL_MIN, LDBL_MAX);
34
35     // Pointer sizes
36     printf("char* is %d bytes\\n", sizeof(char*));
37     printf("short* is %d bytes\\n", sizeof(short*));
38     printf("int* is %d bytes\\n", sizeof(int*));
39
40     return 0;
41 }

```

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	$2^{64} - 1$
signed 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. Go to *Visual Studio 2019* on the start

menu, and then choose *x64 Native Tools Command Prompt*. From here, it is just the same process as before. Use `cl` to build the applications just like before.

**On a Mac?** You're already compiling in 64 bit mode with `clang` by default.

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:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(int argc, char **argv)
6 {
7     // Declare some strings
8     char *hello = "Hello World!";
9     char goodbye[15] = "Goodbye World!";
10
11     // Print the messages
12     printf("Hello message: %s\n", hello);
13     printf("Goodbye message: %s\n", goodbye);
14
15     // Get their size
16     printf("Size of hello message: %d bytes\n", sizeof(hello));
17     printf("Size of goodbye message: %d bytes\n", sizeof(goodbye));
```

```
18
19 // Get the size of the data pointed to by the string
20 printf("Size of data in hello message: %d bytes\n", sizeof(*
    hello));
21 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 from a `int` to a `float` we would do the following:

```
1 int x = 10;
2 float y = (float)x;
3 // 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:

```
1 int x = 10;
2 char *y = (char*)x;
3 // 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.

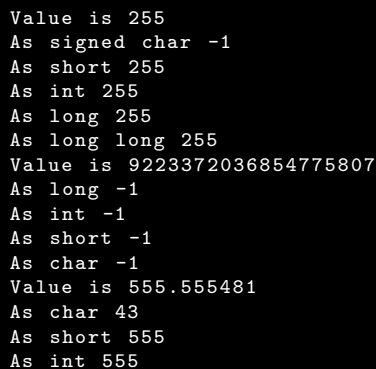
Casting is a useful operation when working with different number types. However, it does have its limitations as our test application will explore. In particular you need to think about what happens when you take a large number (say a `long` long) and convert it to a shorter number (say a `short`). What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

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

```
4
5 int main(int argc, char **argv)
6 {
7     // Let's start with a unsigned char
8     unsigned char c = UCHAR_MAX;
9     printf("Value is %d\n", c);
10    // Cast to signed char
11    printf("As signed char %d\n", (char)c);
12    // Cast to a short
13    printf("As short %d\n", (short)c);
14    // Cast to a int
15    printf("As int %d\n", (int)c);
16    // Cast to a long
17    printf("As long %ld\n", (long)c);
18    // Cast to long long
19    printf("As long long %lld\n", (long long)c);
20
21    // Now try a long long
22    long long l = LLONG_MAX;
23    printf("Value is %lld\n", l);
24    // Cast to long
25    printf("As long %ld\n", (long)l);
26    // Cast to int
27    printf("As int %d\n", (int)l);
28    // Cast to short
29    printf("As short %d\n", (short)l);
30    // Cast to char
31    printf("As char %d\n", (char)l);
32
33    // Now with a float
34    float f = 555.5555f;
35    printf("Value is %f\n", f);
36    // Cast to char
37    printf("As char %d\n", (char)f);
38    // Cast to short
39    printf("As short %d\n", (short)f);
40    // Cast to int
41    printf("As int %d\n", (int)f);
42
43    return 0;
44 }
```

Listing 2.12: Casting Between Types

Running this application will give you the following output:

A terminal window with a black background and white text showing the output of the program. The output consists of 17 lines, each starting with a label followed by a value. The labels are: 'Value is', 'As signed char', 'As short', 'As int', 'As long', 'As long long', 'Value is', 'As long', 'As int', 'As short', 'As char', 'Value is', 'As char', 'As short', and 'As int'. The values are: 255, -1, 255, 255, 255, 255, 9223372036854775807, -1, -1, -1, -1, 555.555481, 43, 555, and 555.

```
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:

$$\textit{float} \Rightarrow \textit{int} \Rightarrow \textit{char}$$

Or in binary:

$$\text{binary float} \Rightarrow 0000\ 0000\ 0000\ 0000\ 0000\ 0010\ 0010\ 1011 \Rightarrow 0010\ 1011$$

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:

```
1 struct student
2 {
3     unsigned int matric;
4     char *name;
5     char *address;
6 };
```

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.



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

### 2.8.2 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.

### 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.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // A data structure containing student information
5 struct student
6 {
```

```
7   unsigned int matric;
8   char *name;
9   char *address;
10 };
11
12 // Prints student information
13 void print_student(struct student s)
14 {
15     printf("matric no: %u\n", s.matric);
16     printf("name: %s\n", s.name);
17     printf("address: %s\n", s.address);
18 }
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:

```

1 enum CHOICE
2 {
3     EXIT = 0,
4     HELLO = 1,
5     GOODBYE = 2
6 };

```

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:

```

1 CHOICE ch;

```

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

```

1 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:

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 // Defines the menu choices
6 enum CHOICE
7 {
8     EXIT = 0,
9     HELLO = 1,
10    GOODBYE = 2
11 };
12
13 int main(int argc, char **argv)
14 {
15     int flag = 1;
16     while (flag)
17     {
18         // Print menu
19         printf("1 - say hello\n");
20         printf("2 - say goodbye\n");
21         printf("0 - exit\n");
22         printf("Enter choice - ");
23         // Read input
24         char buffer[10];
25         fgets(buffer, 10, stdin);
26         // Convert to CHOICE
27         enum CHOICE ch = atoi(buffer);
28
29         // Work on input
30         switch (ch)
31         {
32             case HELLO:
33                 printf("Hello World!\n");
34                 break;
35             case GOODBYE:
36                 printf("Goodbye World!\n");
37                 break;
38             case EXIT:

```

```
39         printf("Exiting...\n");
40         flag = 0;
41         break;
42     default:
43         printf("*** INVALID INPUT ***\n");
44         break;
45     }
46 }
47
48 return 0;
49 }
```

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.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // Constant global value
5 const int x = 500;
6
7 void func(const int i)
8 {
9     // Print the constant
```

```

10     printf("i = %d\n", i);
11     // Change value of i - compiler error
12     i = 50;
13 }
14
15 int main(int argc, char **argv)
16 {
17     // Print the constant
18     printf("x = %d\n", x);
19     // 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.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3
4 void func()
5 {
6     // Define static value
7     static int x = 0;
8
9     // Print value of x
10    printf("x = %d\n", x);
11    // Increment x
12    x = x + 1;
13 }
14

```

```
15 int main(int argc, char **argv)
16 {
17     // Call func 100 times
18     for (int i = 0; i < 100; ++i)
19         func();
20
21     return 0;
22 }
```

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:

```
1 hello:
2     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.

**On a Mac?** Makefiles work in exactly the same way. The only difference is that the command is **make** instead of **nmake**.

## 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.

```

1 all: sizeof unsigned minmax datatypes
3 sizeof:
4     cl sizeof.c
6 unsigned:
7     cl unsigned.c
9 minmax:
10    cl minmax.c
12 datatypes:
13    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:

```
1 clean:
2     del *.obj
3     del *.exe
4     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, with a focus on the close correspondance between lines of C and lines of assembly language.

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:

C code  $\Rightarrow$  Assembly code  $\Rightarrow$  Object code  $\Rightarrow$  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.

For this unit, you will need to use the 32 bit Microsoft compiler (the standard Developer Command Prompt for VS 2019), as we are using 32 bit Intel assembly.

**On a Mac?** For this unit only, you will need to use the Microsoft compiler on Windows to follow along. If you do not already have Windows installed (e.g. through Bootcamp or on a virtual machine) then you can access Visual Studio 2019 through Apps Anywhere on the University's Virtual Desktop service (<https://desktop.napier.ac.uk>), which avoids you needing to install Windows and Visual Studio on your machine. You can also access Notepad++ through Apps Anywhere on the Virtual Desktop service to type in your source code.

### 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:

```
1  __asm
2  {
3      // Assembly commands go here
4  }
```

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 <code>if</code> and <code>while</code> .
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:

```
1 x = y;
```

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

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv)
4 {
5     int x = 500;
6     int y = 0;
7
8     printf("Before assembly, x = %d, y = %d\n", x, y);
9
10    __asm
11    {
12        mov eax, x
13        mov y, eax
14    }
15
16    printf("After assembly, x = %d, y = %d\n", x, y);
17    return 0;
18 }
```

Listing 3.1: First Application Using Inline Assembly

For this application we want to write the assembly code that the compiler generates to a file that we can then view (with the default compilation settings the assembly code is only held in memory and is immediately assembled into object code before being discarded). To do this we use `cl` as follows:

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

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

```
1 ; Listing generated by Microsoft (R) Optimizing Compiler Version 18
2   .00.30723.0
3
4   TITLE E:\Repos\programming-fundamentals\code\unit-03\first-
5     inline.c
6   .686P
7   .XMM
8   include listing.inc
9   .model flat
10
11  INCLUDELIB LIBCMT
12  INCLUDELIB OLDNAMES
13
14  _DATA SEGMENT
15  $SG3050 DB 'Before assembly, x = %d, y = %d', 0aH, 00H
16  ORG $+3
17  $SG3051 DB 'After assembly, x = %d, y = %d', 0aH, 00H
18  _DATA ENDS
19  PUBLIC _main
20  EXTRN _printf:PROC
21  ; Function compile flags: /Odtp
22  _TEXT SEGMENT
23  _x$ = -8 ; size = 4
```

```
22 _y$ = -4           ; size = 4
23 _argc$ = 8         ; size = 4
24 _argv$ = 12        ; size = 4
25 _main PROC
26 ; File e:\repos\programming-fundamentals\code\unit-03\first-
    inline.c
27 ; Line 4
28 push ebp
29 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.

Running the application `.exe` 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 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.

```

1 #include <stdio.h>
2
3 int main(int argc, char **argv)
4 {
5     int x = 0;
6
7     printf("Before assembly, x = %d\n", x);
8
9     __asm
10    {
11        // Same as x = 500
12        mov x, 500
13    }
14
15    printf("After assembly, x = %d\n", x);
16
17    return 0;
18 }

```

Listing 3.3: Second Inline Assembly Application

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

```

1 x = 500;

```

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

```

1 ; Listing generated by Microsoft (R) Optimizing Compiler Version 18
   .00.30501.0
2
3 TITLE D:\programming-fundamentals\code\unit-03\second-inline.c
4 .686P
5 .XMM
6 include listing.inc
7 .model flat
8
9 INCLUDELIB LIBCMT
10 INCLUDELIB OLDNAMES
11
12 _DATA SEGMENT
13 $SG3049 DB 'Before assembly, x = %d', 0aH, 00H
14     ORG $+3
15 $SG3050 DB 'After assembly, x = %d', 0aH, 00H
16 _DATA ENDS
17 PUBLIC _main
18 EXTRN _printf:PROC
19 ; Function compile flags: /Odtp
20 _TEXT SEGMENT
21 _x$ = -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 are 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.

```
1 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:

```
1 void func()  
2 {  
3     // Enter the function  
4     // Declare 2 values  
5     int x = 500;  
6     int y = 1000;  
7     {  
8         // Enter an inner scope  
9         int z = x + y;  
10        // Exit inner scope - z removed from stack  
11    }  
12    // Exit the function - x and y removed from stack  
13 }
```

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.

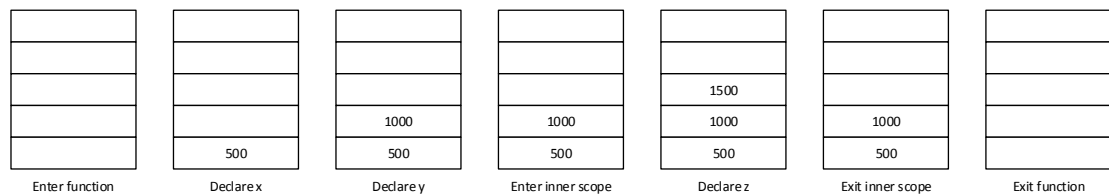


Figure 3.1: Values Added and Removed from the Stack During Operation

#### 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.

```

1 #include <stdio.h>
2
3 int main(int argc, char **argv)
4 {
5     int x = 500;
6     int y = 200;
7
8     __asm
9     {
10         // Push x onto the stack
11         push x
12         // Push y onto the stack
13         push y
14         // Pop stack into x
15         pop x
16         // Pop stack into y
17         pop y
18     }
19
20     printf("x = %d, y = %d\n", x, y);
21
22     return 0;
23 }

```

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.

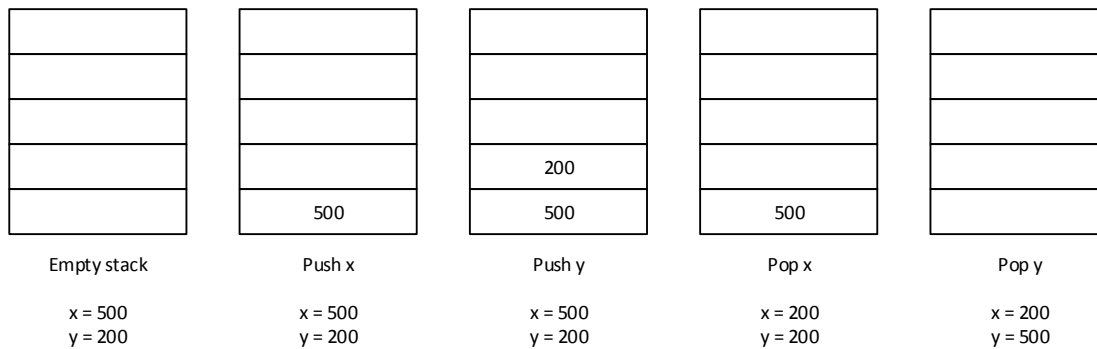


Figure 3.2: Using the Stack to Swap Values with push and pop

```
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 its parameter list), we have to push these values onto the stack.

As an example, consider a function defined as follows:

```
1 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:

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

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

```
1 push 2000
2 push 1000
3 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.

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv)
4 {
5     char *message = "Hello World!\n";
6
7     __asm
8     {
9         // Push message onto the stack
10        push message
11        // Call printf
12        call printf
13        // Clean up stack - pop into ebx
14        pop ebx
15    }
16
17    return 0;
18 }
```

Listing 3.10: Calling printf From Assembly

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

```
1 ; Listing generated by Microsoft (R) Optimizing Compiler Version 18
2   .00.30501.0
3
4     TITLE D:\programming-fundamentals\code\unit-03\call.c
5     .686P
6     .XMM
7     include listing.inc
8     .model flat
9
10    INCLUDELIB LIBCMT
11    INCLUDELIB OLDNAMES
12
13    _DATA SEGMENT
14    $SG3049 DB 'Hello World!', 0aH, 00H
15    _DATA ENDS
16    PUBLIC _main
17    EXTRN _printf:PROC
18    ; Function compile flags: /Odtp
19    _TEXT SEGMENT
20    _message$ = -4 ; size = 4
21    _argc$ = 8 ; size = 4
22    _argv$ = 12 ; size = 4
23    _main PROC
24    ; File d:\programming-fundamentals\code\unit-03\call.c
25    ; Line 4
26    push ebp
27    mov ebp, esp
28    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:

```
1 int z = subtract(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 subtract
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.

```
1 #include <stdio.h>
2
3 int subtract(int x, int y)
4 {
5     return x - y;
6 }
7
8 int main(int argc, char **argv)
9 {
10     int result = 0;
11
12     __asm
13     {
14         // Push 500 onto stack
15         push 500
16         // Push eax onto stack
17         push 200
18         // Call subtract
19         call sub
20         // Move eax into result
21         // eax contains the result of the call
22         mov result, eax
23         // Clean up stack - pop into ebx
24         pop ebx
25         pop ebx
26     }
27
28     printf("result = %d\n", result);
29
30     return 0;
31 }
```

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 is the generated assembly code which is below:

```

1 ; Listing generated by Microsoft (R) Optimizing Compiler Version 18
  .00.30501.0
2
3 TITLE D:\programming-fundamentals\code\unit-03\call2.c
4 .686P
5 .XMM
6 include listing.inc
7 .model flat
8
9 INCLUDELIB LIBCMT
10 INCLUDELIB OLDNAMES
11
12 _DATA SEGMENT
13 $SG3055 DB 'result = %d', 0aH, 00H
14 _DATA ENDS
15 PUBLIC _sub
16 PUBLIC _main
17 EXTRN _printf:PROC
18 ; Function compile flags: /OdtP
19 _TEXT SEGMENT
20 _result$ = -4 ; size = 4
21 _argc$ = 8 ; size = 4
22 _argv$ = 12 ; size = 4
23 _main PROC
24 ; File d:\programming-fundamentals\code\unit-03\call2.c
25 ; Line 9
26 push ebp
27 mov ebp, esp
28 push ecx
29 push ebx
30 ; Line 10
31 mov DWORD PTR _result$[ebp], 0
32 ; Line 15
33 push 500 ; 000001f4H
34 ; Line 17
35 push 200 ; 000000c8H
36 ; Line 19
37 call _subtract
38 ; Line 22
39 mov DWORD PTR _result$[ebp], eax
40 ; Line 24
41 pop ebx
42 ; Line 25
43 pop ebx
44 ; Line 28
45 mov eax, DWORD PTR _result$[ebp]
46 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 _subtract 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 **subtract** 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 **subtract** 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:

```

1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main(int argc, char **argv)
5 {
6     // Format string for message
7     char *format = "Argument %d: %s\n";
8     // Final message
9     char *message = "All arguments printed.\n";
10    // Iteration value
11    int i;
12    __asm
13    {
14        // Set i to 0
15        mov i, 0
16        // Jump to check
17        jmp $CHECK_i
18
19        // This section of code increments i
20        // Look up the inc instruction - simplify your code
21        $INCREMENT_i:
22        mov eax, i
23        add eax, 1
24        mov i, eax
25    }

```

---

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

---

```

1: procedure ASSEMBLY_LOOP
2:    $i \leftarrow 0$ 
3:   goto CHECK_i
4:                                     ▷ INCREMENT_i
5:    $eax \leftarrow i$ 
6:    $eax \leftarrow eax + 1$ 
7:    $i \leftarrow eax$ 
8:                                     ▷ CHECK_i
9:    $ecx \leftarrow i$ 
10:  Compare  $ecx$  to  $argc$ 
11:  if greater than then goto END_LOOP
12:   $edx \leftarrow i$ 
13:   $eax \leftarrow argv$ 
14:   $ecx \leftarrow eax + edx \times 4$ 
15:  push  $ecx$  onto the stack
16:  push  $i$  onto the stack
17:  push  $format - message$  onto the stack PRINTF
18:  pop the stack into  $ebx$ 
19:  pop the stack into  $ebx$ 
20:  pop the stack into  $ebx$ 
21:  goto INCREMENT_i
22:                                     ▷ END_LOOP
23:  push  $end - message$  onto the stack PRINTF
24:  pop the stack into  $ebx$ 

```

---

```

26  // This section of code checks if i < argc
27  $CHECK_i:
28      mov ecx, i
29      cmp ecx, argc
30      jge $END_LOOP
31
32  // This is the body of the loop
33  mov edx, i
34  mov eax, argv
35  // This line takes the base address of argv and adds 4 * i
   // bytes further along
36  // Remember that a memory address in 32-bit applications is
   // 4 bytes
37  mov ecx, [eax + (edx * 4)]
38  // Push the arguments onto the stack
39  push ecx
40  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. 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.
3. **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.
4. **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 to 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:

pre-processor  $\Rightarrow$  compiler  $\Rightarrow$  linker

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

Original C  $\Rightarrow$  Pre-processed C  $\Rightarrow$  Assembly  $\Rightarrow$  Object Code  $\Rightarrow$  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.

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

Listing 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.

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

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

Table 4.1: Conditional Compilation

Pre-processor Command	Description
<code>#include</code>	Includes (adds) a code from a header file to the code file as part of the code to be compiled
<code>#define</code>	Defines a value which is then replaced in the code file when found, or used for conditional compilation
<code>#undef</code>	Undefines a value. Can overwrite a <code>#define</code> used previously.
<code>#if</code>	Used to check a value of a defined pre-processor value.
<code>#ifdef</code>	Used to check if a value has been defined
<code>#ifndef</code>	Used to check if a value has not been defined
<code>#else</code>	Used with <code>#if</code> and <code>#ifdef</code>
<code>#elif</code>	An <i>else-if</i> statement
<code>#endif</code>	Ends a pre-processor if block
<code>#pragma</code>	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 an 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++.

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

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

Listing 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.

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

Listing 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:

```
1 #ifndef HELLO_HEADER_GUARD
2 #define HELLO_HEADER_GUARD
3
4 // Code defined here...
5
6 #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.

```
1 // Include the hello header. Note the use of quotes this time
2 #include "hello.h"
3
4 int main(int argc, char **argv)
5 {
6     hello_world();
7 }
```

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 `c1` 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:

```
1 #include <stdio.h>
2
3 void hello_world()
4 {
5     printf("Hello world!\n");
6 }
7
8 int main(int argc, char **argv)
9 {
10     hello_world();
11 }
```

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 can 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, an 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.

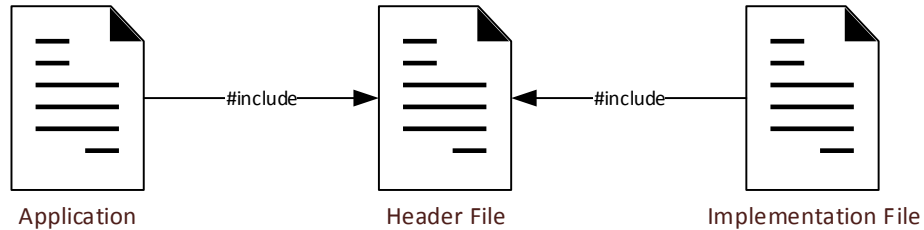


Figure 4.1: Inclusion Structure

### 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:

```

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

Listing 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:

```

1 #include "student.h"
2 #include <stdio.h>
3
4 void print_student(struct student s)
5 {
6     printf("Matric: %d\n", s.matric);
7     printf("Name: %s\n", s.name);
8     printf("Address: %s\n", s.address);
9 }
  
```

Listing 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:

```

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

```

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.

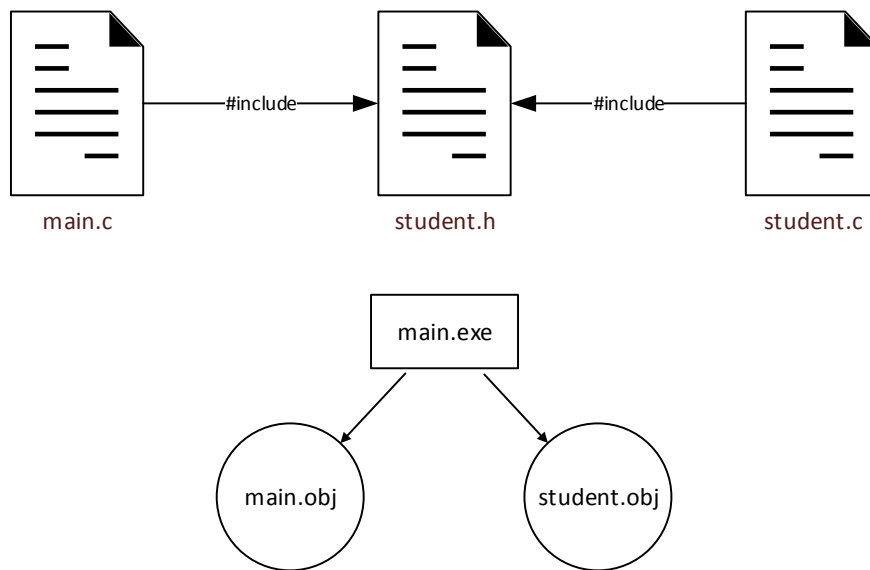


Figure 4.2: Structure of Student Application - both File Inclusion and Object Files

### 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.

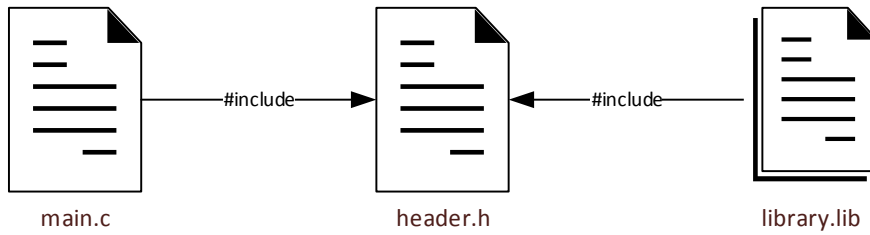


Figure 4.3: Library Inclusion Structure

```
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.

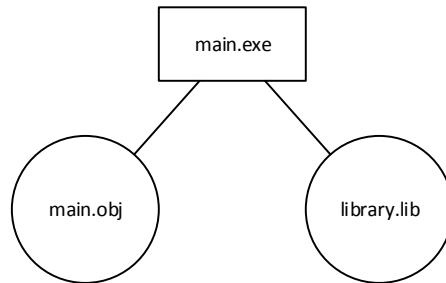


Figure 4.4: Linking a Library

### 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 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.

```

1 #pragma once
2
3 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

---

```

1: function SEARCH(value, size, data)
2:   for  $i \leftarrow 0$  to  $size - 1$  do
3:     if  $data_i = value$  then
4:       return  $i$ 
5:   return  $-1$ 

```

---

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

```

1 #include "search.h"
2
3 int search(int value, int size, int *data)
4 {
5     // Loop through data until found
6     for (int i = 0; i < size; ++i)
7     {
8         if (data[i] == value)
9         {
10             // Found value - return i
11             return i;
12         }
13     }
14     // Not found. Return -1
15     return -1;
16 }

```

Listing 4.17: Main Search File

### 4.5.3 sort.h

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

1. the size of the array to be sorted

2. the array to be sorted - as a pointer

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

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

Listing 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 their 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

---

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

---

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

```
1 #include "sort.h"
2 #include <stdio.h>
3
4 void sort(int size, int *data)
5 {
6     // Iterate through each value
7     for (int i = 0; i < size; ++i)
8     {
9         // Loop through values above index i
10        for (int j = 0; j < size - (i + 1); ++j)
11        {
12            // Test if data[j] > data[j + 1]
13            if (data[j] > data[j + 1])
14            {
15                // Swap values
16                int temp = data[j + 1];
17                data[j + 1] = data[j];
18                data[j] = temp;
19            }
20        }
21        // Display % of currently sorted data
22        if (i % 1000 == 0)
23            printf("%.2f%% sorted\n", ((float)i / (float)size) * 100.0f);
24    }
25 }
```

Listing 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.

```
1 #pragma once
2
3 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.

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

Listing 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 *Seed Random*. 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`.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include "generate.h"
4 #include "sort.h"
5 #include "search.h"
6
7 #define NUM_INTEGERS 65535
8
9 int main(int argc, char **argv)
10 {
11     // Allocate an array of a given size
12     int data[NUM_INTEGERS];
13
14     // Generate random numbers
15     generate(NUM_INTEGERS, data);
16
17     // Output first 20 values
18     printf("\nUnsorted\n");
19     for (int i = 0; i < 20; ++i)
20         printf("%d\n", data[i]);
21
22     // Sort the data
23     sort(NUM_INTEGERS, data);
24
25     // Output first 20 values
26     printf("\nSorted\n");
27     for (int i = 0; i < 20; ++i)
28         printf("%d\n", data[i]);
29
30     return 0;
31 }
```

Listing 4.21: Test Application for Array Library

Our inclusion file structure is shown in Figure 4.5

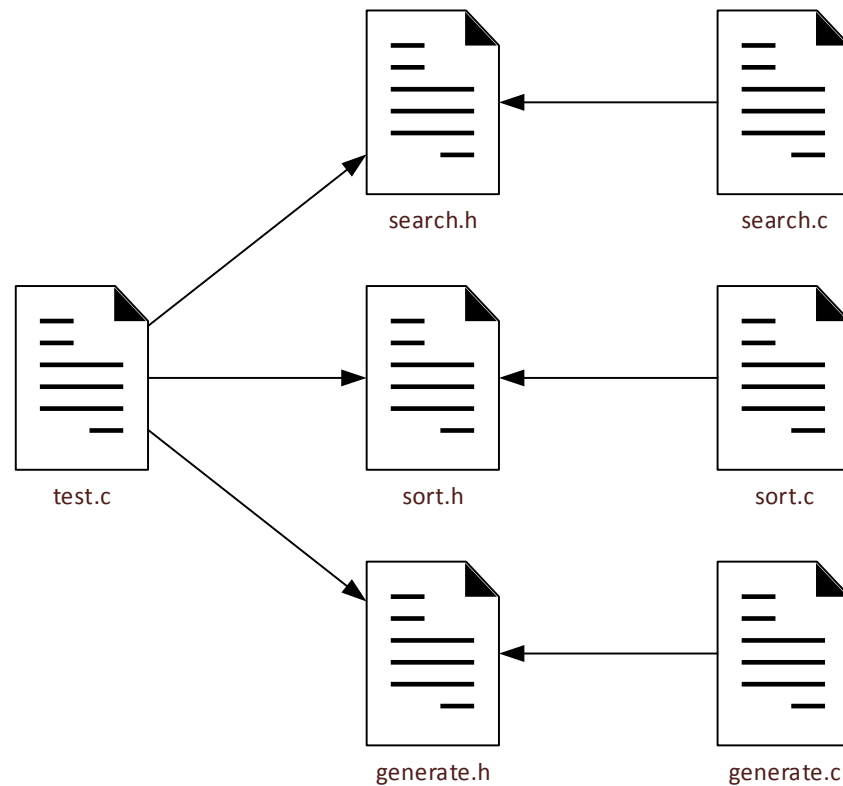


Figure 4.5: Array Library File Structure

### 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:

input  $\Rightarrow$  process  $\Rightarrow$  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.

```
1 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 Reading Text Files

**This section is important for your coursework!** To read in text files, we can use `fgets` to read in a line at a time, once we've opened the file with `fopen`. Recall that we have been entering `stdin` as the third argument to `fgets`, to read in from the keyboard (standard input). In fact, `stdin` is of type `FILE*`. This means that we can use any `FILE*` as the third argument to `fgets`, for example, we can use a text file opened with `fopen`. Each call to `fgets` will then read in the next line of the file, storing it in the array of characters that we specify as its first argument. When it reaches the end of the file, `fgets` returns `NULL`.

The typical pattern to read every line in a text file in turn is to put the call to `gets` inside the condition part of a while loop:

```
1 FILE *input = fopen("input file", "r");
2 char line [1000]; //Will store each line in turn
3 while(fgets(line, 1000, input) != NULL)
4 {
5     //Do something with line, e.g. print it
6     printf("%s\n", line);
7 }
8 fclose(input); //Close the input file when we have finished reading
   it
```

Listing 4.25: Reading every line of a file in turn with `fgets` and printing it.

Let's now write a simple spell checking application. This application will ask the user to enter a word they would like to check the spelling of. It will then read in a dictionary, stored as a text file, with one word on each line. It will compare the word that the user has entered to each word in the dictionary file in turn. If

the words match, then the word the user entered is correctly spelt. If, on the other hand, the end of the dictionary is reached and no match has been found then the word the user entered is not spelt correctly.

The code for the application looks like this:

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main (int argc, char **argv)
5 {
6     FILE *input = fopen("dictionary.txt", "r");
7     //Read in a word to check the spelling of
8     char search_word [100] ;
9     printf("Please enter a word to check the spelling of: ");
10    fgets(search_word, 100, stdin);
11    //remove the newline character captured by fgets
12    int len = strlen(search_word);
13    if(len > 0 && search_word[len - 1] == '\n')
14    {
15        search_word[len - 1] = '\0';
16    }
17    //Read in the dictionary file line by line
18    //Check if it contains search_word
19    char dictionary_line [100];
20    int dict_line_len = 0;
21    int found = 0; //not found the word in the dictionary yet
22    while(fgets(dictionary_line, 100, input) != NULL){
23        //Need to remove the new line at the end of the word
24        dict_line_len = strlen(dictionary_line);
25        if(dict_line_len > 0 && dictionary_line[dict_line_len - 1]
26           == '\n')
27        {
28            dictionary_line[dict_line_len - 1] = '\0';
29        }
30        if(strcmp(search_word, dictionary_line) == 0)
31        {
32            //Strings match
33            printf("Word %s is spelt correctly\n", search_word);
34            found = 1;
35            break;
36        }
37    }
38    if(found == 0)
39    {
40        printf("Word %s is not in the dictionary\n", search_word);
41    }
42    fclose(input);
43    return 0;
44 }
```

Listing 4.26: A simple spell checker application.

You should compile this code and run it. You will find the `dictionary.txt` file in the Practical Materials section on Moodle. Test it with different input words.

**Exercise:** Modify the spell checker code so that it is case insensitive, e.g. “Hello” matches as well as “hello”.

### 4.6.3 Tokenising (splitting) strings

A common requirement is to be able to split a string into separate parts, or *tokens*. For example, you may wish to read in a line that contains an English sentence, and split that sentence into its separate words so that each word can be processed separately. A *token* is defined as a sequence of characters that ends with a *delimiter* character, such as a space or new line.

The code below illustrates how to split a single line of text into its component words, where a word is here defined as a sequence of characters that ends with a space or a newline. In this code, we read in a line of text with `fgets`. We then loop over this line character by character, copying the characters into an array called `token`. We stop copying when we get to a space, add a null terminator onto the end of `token` to make it a proper null terminated string, and then print it. This is the basic strategy to process a line of text in C: copy the characters into an array that will store one token, stop copying when you get to a space or other delimiter, add a null terminator onto the end, and then process the token (e.g. print it, or check if it matches a word in a dictionary).

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main (int argc, char **argv)
5 {
6     char sentence [1000];
7     printf("Please enter a sentence:\n");
8     fgets(sentence, 1000, stdin);
9     // Loop along the characters of the line until we find a space
10    // or newline,
11    // copying the characters into token, then print the token
12    int sentence_length = strlen(sentence);
13    char token [100]; // Will store a single token (word)
14    int token_index; // Will store where we are in the token array
15    int sentence_index = 0;
16    while(sentence_index < sentence_length &&
17           sentence[sentence_index] != '\n')
18    {
19        //Get the next token (word)
20        token[0] = '\0'; // Reset token by setting first character
21                          // to '\0';
22        token_index = 0; // Reset the token index
23        while(sentence[sentence_index] != ' ' &&
24               sentence[sentence_index] != '\n')
25        {
26            // Copy the character into token
27            token[token_index] = sentence[sentence_index];
28            sentence_index++; // Move along one in the sentence
29            token_index++; // Move along one in the token
30        }
31        // The token ends here
32        token[token_index] = '\0'; // Insert a '\0' to mark the end
33        printf("%s\n", token); //Print the token
34        sentence_index++; // Move past the space in the sentence
35    }
36    return 0;
37 }

```

Listing 4.27: Example code to split a sentence into separate words.

**Exercise:** Modify the word splitting code so that it removes commas and full stops from the end of a word. You should do this by including commas and full stops as token delimiters, alongside spaces and newlines.

To make life easier in some situations, you can also use the `strtok` function from the standard library. The code example below illustrates this. The `strtok` function takes in as arguments the string that you wish to tokenise, and a string of *delimiters* – characters that mark the end of a token. Each call to `strtok` returns the next token. Note that after the first call to `strtok`, which includes the string to tokenise, subsequent calls pass in `NULL` (this causes `strtok` to continue working along the same string).

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main (int argc, char **argv)
5 {
6     char sentence [1000]; // Array to store the sentence we're
7     // reading in
8     printf("Please enter a sentence:\n");
9     fgets(sentence, 1000, stdin);
10    char *delimiters = " .,\n"; // Symbols that mark the end of a
11    // token (word)
12    char *token = strtok(sentence, delimiters); // Get the first
13    // token
14    while(token != NULL) // Loop until there are no more tokens (
15    // words) left
16    {
17        printf("%s\n", token); // Print the token
18        token = strtok(NULL, delimiters); //Get the next token (
19        // note the NULL as the first arg)
20    }
21    return 0;
22 }
```

Listing 4.28: Using the standard library `strtok` function to tokenise a string.

#### 4.6.4 Reading a Binary File

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

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

Listing 4.29: 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.



### 4.6.5 Reading a Binary File

Let us now write a test application to open a binary file, read it in, and then sort it. To do this we will also write a function that will read in a file and return the amount of data read. The form of the data file will be such that the first 4 bytes will tell us the number of values stored in the data file. This means that we don't know how many values are stored, so we will have to introduce some strategies for allocating enough space to store our values. The code for our test application is below. We will explain the new parts presently.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include "sort.h"
4
5 // Reads in a block of data as an int array
6 int readfile(int **data)
7 {
8     // Open file for reading
9     FILE *file;
10    file = fopen("numbers.dat", "rb");
11    // First value is number of integers
12    int size;
13    fread(&size, sizeof(int), 1, file);
14    // Allocate memory for that number of values
15    *data = (int*)malloc(sizeof(int) * size);
16    // Read in rest of data
17    fread(*data, sizeof(int), size, file);
18    //Close the file now that we have finished with it
19    fclose(file);
20    // Return size
21    return size;
22 }
23
24 int main(int argc, char **argv)
25 {
26     // Declare data
27     int *data;
28     // Read in file
29     int size = readfile(&data);
30     // Sort
31     sort(size, data);
32     // Print first 20 results
33     for (int i = 0; i < 20; ++i)
34         printf("%d\n", data[i]);
35
36     // Free the allocated memory - otherwise a memory leak! (very
37     // bad)
38     free(data);
39
40     return 0;
41 }

```

Listing 4.30: Reading a Binary File

#### Where is the numbers.dat File?

You will find this on Moodle, in the Practical Materials section.

**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.

#### 4.6.6 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.

```

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

```

```
36
37 int main(int argc, char **argv)
38 {
39     // Read in data
40     int *data;
41     int size = readfile(&data);
42     // Sort
43     sort(size, data);
44     // Write the data
45     writefile(size, data);
46     // Free the allocated memory - otherwise a memory leak! (very
47     //    bad)
48     free(data);
49
50     return 0;
51 }
```

Listing 4.31: 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 it 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 /EHsc hello.cpp`. The `/EHsc` argument tells the compiler about the kind of exception handling to use. We won't be covering exception handling in this module, but without the argument you will get a warning with every program that you build. Again, this is where makefiles are useful!

**On a Mac?** Compile using `clang++ -o hello hello.c`. The `clang++` tells `clang` to link to the C++ standard library.

```

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 insertion* operator. You can read the entire command as *insert into the console stream "Hello World!" then an end line*. There is an equivalent *stream extraction* operator (`>>`) which is used for input from the console, and which you will 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 1.18) in C++. The code is below:

```

1 #include <iostream>
2 #include <string>
3
4 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() << "
16         characters long" << std::endl;
17
18     return 0;
19 }
```

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 1.10. 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 <stdio>
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 [1.10](#) for C but in C++.

```

1 #include <iostream>
2 #include <string>
3
4 int main(int argc, char **argv)
5 {
6     // Declare strings to store name
7     std::string first_name;
8     std::string last_name;
9     std::string full_name;
10
11     // Prompt for first name
12     std::cout << "Please enter your first name: ";
13     // Read in first_name using cin
14     std::cin >> first_name;
15
16     // Check if your name is Kevin
17     if (first_name == "Kevin")
18         std::cout << "Hey! Another Kevin\n";
19     else
20         std::cout << "Oh well\n";
21
22     // Output number of characters entered
23     std::cout << "You entered " << first_name.length() << "
24         characters" << std::endl;
25     // Prompt for last name
26     std::cout << "Please enter your last name: ";
27     std::cin >> last_name;
28     // Output number of characters entered
29     std::cout << "You entered " << last_name.length() << "
30         characters" << std::endl;
31
32     // Join the strings
33     full_name = first_name + std::string(" ") + last_name;
34
35     // Print name

```

```
34     std::cout << "Your full name is " << full_name << " which is ";
35     std::cout << full_name.length() << " characters long" << std::
        endl;;
36
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 extraction (`>>`) 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.

```
1 #include <iostream>
2 #include <string>
3
4 int main(int argc, char **argv)
5 {
6     // Declare string to store name
7     std::string full_name;
8
9     // Prompt for name
10    std::cout << "Please enter your full name: ";
11    // Read in full_name using cin and getline
12    std::getline(std::cin, full_name);
13
14    // Print name
15    std::cout << "Your full name is " << full_name << " which is ";
16    std::cout << full_name.length() << " characters long" << std::
        endl;;
17
18    return 0;
19 }
```

19 }

Listing 5.7: Using `getline`

Line 12 is where we are using the `getline` function. Notice it 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.

```

1 #include <iostream>
2
3 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 5.8: 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

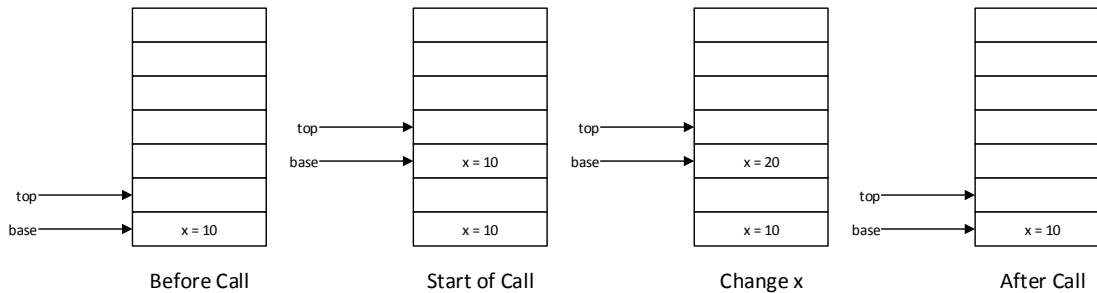


Figure 5.1: Interpretation of the Stack During Pass-by-Value

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:

```
1 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	<code>int c = a &amp; c</code>	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	<code>bool c = a &amp;&amp; b</code>	Performs a logical and on two boolean values.
Address of operator	<code>int *a = &amp;b</code>	Gets the address (to create a pointer) of a value.
Reference to specifier	<code>int &amp;a = b</code>	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.

```

1 #include <iostream>
2
3 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.

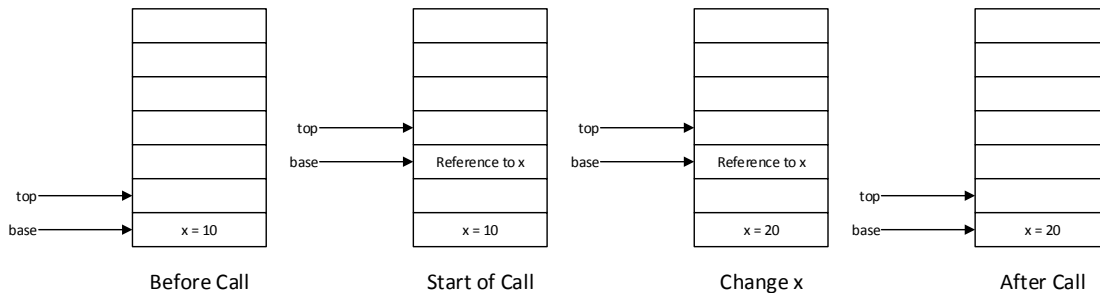


Figure 5.2: Interpretation of the Stack Using Pass-by-Reference

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 its 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 its 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.

```

1 #include <iostream>
2 #include <vector>
3
4 void sort_copy(std::vector<int> data)
5 {
6     // Iterate through each value
7     for (int i = 0; i < data.size(); ++i)
8     {
9         // Loop through values above index i
10        for (int j = 0; j < data.size() - (i + 1); ++j)
11        {
12            // Test if data[j] > data[j + 1]
13            if (data[j] > data[j + 1])
14            {
15                // Swap values
16                int temp = data[j + 1];
17                data[j + 1] = data[j];
18                data[j] = temp;
19            }
20        }
21        if (i % 1000 == 0)
22            std::cout << ((float)i / (float)data.size()) * 100.0f << "%
                sorted" << std::endl;
23    }
24 }
25
26 void sort_reference(std::vector<int> &data)
27 {
28     // Iterate through each value
29     for (int i = 0; i < data.size(); ++i)
30     {
31         // Loop through values above index i
32         for (int j = 0; j < data.size() - (i + 1); ++j)
33         {
34             // Test if data[j] > data[j + 1]
35             if (data[j] > data[j + 1])
36             {
37                 // Swap values
38                 int temp = data[j + 1];
39                 data[j + 1] = data[j];
40                 data[j] = temp;
41             }
42         }
43         if (i % 1000 == 0)
44             std::cout << ((float)i / (float)data.size()) * 100.0f
45                 << "% sorted" << std::endl;
46    }
47 }
48 int main(int argc, char **argv)
49 {
50     std::vector<int> data;
51     for (int i = 0; i < 262144; ++i)
52         data.push_back(262144 - i);

```



```

53 |
54 |     std::cout << "Sorting by copy..." << std::endl;
55 |     sort_copy(data);
56 |     std::cout << "Sorting by reference..." << std::endl;
57 |     sort_reference(data);
58 |
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<sup>th</sup> element, for the second call the 1<sup>st</sup> 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

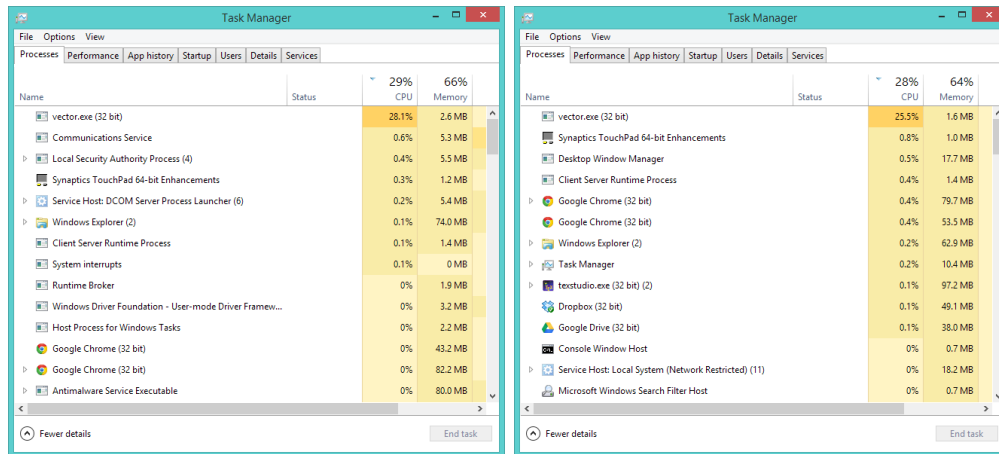


Figure 5.3: Task Manager During Pass-by-Copy (left) and Pass-by-Reference (right)

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.

### 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.

```

1 #pragma once
2
3 #include <vector>
4
5 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.

```

1 #include "sort.h"
2 #include <iostream>
3
4 void sort(std::vector<int> &data)
5 {
6     // Iterate through each value
7     for (int i = 0; i < data.size(); ++i)
8     {
9         // Loop through values above index i
10        for (int j = 0; j < data.size() - (i + 1); ++j)
11        {
12            // Test if data[j] > data[j + 1]
13            if (data[j] > data[j + 1])
14            {

```

```

15         // Swap values
16         int temp = data[j + 1];
17         data[j + 1] = data[j];
18         data[j] = temp;
19     }
20 }
21 if (i % 1000 == 0)
22     std::cout << ((float)i / (float)data.size()) * 100.0f
23     << "% sorted" << std::endl;
24 }

```

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:

```

1 #pragma once
2
3 #include <vector>
4
5 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.

```

1 #include "generate.h"
2 #include <random>
3
4 std::vector<int> generate(int size)
5 {
6     // Create random generator
7     std::random_device rd;
8     // Create distribution
9     std::uniform_int_distribution<int> dist;
10    // Generate random numbers
11    std::vector<int> data;
12    for (int i = 0; i < size; ++i)
13        data.push_back(dist(rd));
14
15    return data;
16 }

```

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.

```
1 #include "generate.h"
2 #include "sort.h"
3 #include <iostream>
4
5 int main(int argc, char **argv)
6 {
7     std::vector<int> data = generate(65536);
8     sort(data);
9
10    // Print the first 100 values
11    for (int i = 0; i < 100; ++i)
12        std::cout << data[i] << std::endl;
13
14    return 0;
15 }
```

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
<code>istream</code>	input stream	A stream that data can be read from.
<code>ostream</code>	output stream	A stream that data can be written to.
<code>iostream</code>	input-output stream	A stream that can be read from and written to
<code>fstream</code>	file stream	Represents an input-output stream that points to a file. <code>ifstream</code> and <code>ofstream</code> also exist.
<code>stringstream</code>	string stream	Provides a stream that can be written to to create a <code>string</code> . 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 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 beginning 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 two 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.

```
1 #include <iostream>
2 #include <fstream>
3 #include <vector>
4 #include <string>
```

```

5
6 std::vector<std::string> readfile(std::string filename)
7 {
8     // Open file - default is text
9     std::ifstream file(filename);
10    // String to read into
11    std::string line;
12    // 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 {
26     // Open file - default is text
27     std::ofstream file(filename);
28     // Write each line into file starting at end of vector
29     for (int i = data.size() - 1; i >= 0; --i)
30         file << data[i] << std::endl;
31     // Close the file
32     file.close();
33 }
34
35 int main(int argc, char **argv)
36 {
37     // Read in file
38     std::vector<std::string> data = readfile(std::string("sorted.
39         txt"));
40     // Print lines read
41     std::cout << "Lines read = " << data.size() << std::endl;
42     writefile(std::string("reversed.txt"), data);
43
44     return 0;
45 }

```

Listing 5.20: Writing Text Files in C++

Line 15 uses the `getline` 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.

```

1 #include <iostream>
2 #include <string>
3

```



```

4 std::string join(const std::string &a, const std::string &b)
5 {
6     // This line won't compile
7     return a.append(b);
8 }
9
10 int main(int argc, char **argv)
11 {
12     std::string greeting = join(std::string("Hello,"), std::string(
13         " World!"));
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.

```

1 std::string join(const std::string &a, const std::string &b)
2 {
3     // This line will compile
4     return a + b;
5 }

```

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. One case where you must use `const` is if you wish to pass a string literal, such as `"input.txt"` into a function by reference. In this case you must mark the parameter as a `const` reference in the function, to prevent the function from changing it.

### 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 were 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

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

Table 5.2: Pointer Operators and Specifiers

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.

```

1 #include <iostream>
2
3 void foo(int *x)
4 {
5     std::cout << "Address of x in function = " << x << std::endl;
6     std::cout << "Start of function, x = " << *x << std::endl;
7     // Have to dereference pointer to change value
8     *x = 20;
9     std::cout << "End of function, x = " << *x << std::endl;
10    std::cout << "Address of x at end of function = " << x << std::
        endl;
11 }
12
13 int main(int argc, char **argv)
14 {
15     int x = 10;
16     std::cout << "Starting address of x = " << &x << std::endl;
17     std::cout << "Before function call, x = " << x << std::endl;
18     // Have to pass the pointer (or address of) x to the function
19     foo(&x);
20     std::cout << "After function call, x = " << x << std::endl;
21     std::cout << "End address of x = " << &x << std::endl;
22
23     return 0;
24 }

```

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:

```
1 #include <iostream>
2
3 int* create_array()
4 {
5     // This memory is created on the stack
6     int data[20];
7     for (int i = 0; i < 20; ++i)
8         data[i] = i;
9     return data;
10    // Stack emptied - memory gone
11 }
12
13 int* create_array_new()
14 {
15     // Memory created on the heap
16     int *data = new int[20];
17     for (int i = 0; i < 20; ++i)
18         data[i] = i;
19     return data;
```

```

20 // Memory on the heap still relevant
21 }
22
23 void create_array(int *data)
24 {
25     for (int i = 0; i < 20; ++i)
26         data[i] = i;
27 }
28
29 int main(int argc, char **argv)
30 {
31     int *data = create_array();
32     // Print out all elements
33     for (int i = 0; i < 20; ++i)
34         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.

## 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

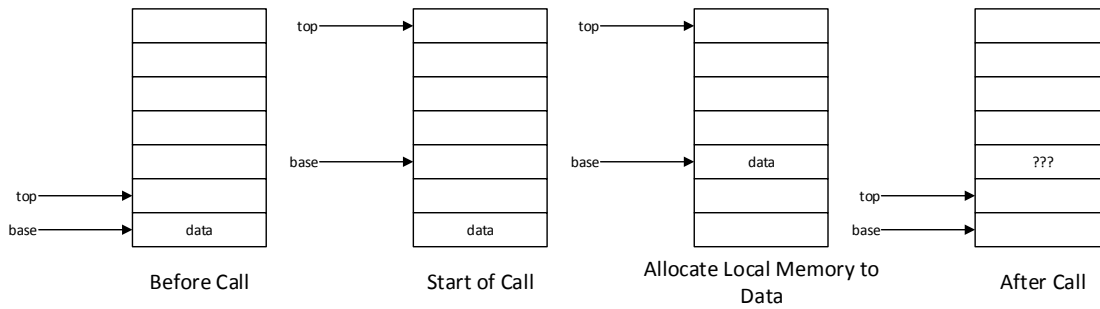


Figure 5.4: Data and Stack Corruption Example - Returning Non-Valid Memory

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

Table 5.3: constness of Pointers

lines you will have to comment out, but you should first build the code without the commented out lines. This is so you can see the compiler error that C++ will provide.

```

1 #include <iostream>
2
3 void foo(const int *x)
4 {
5     std::cout << "Address of x in function = " << x << std::endl;
6     std::cout << "Start of function, x = " << *x << std::endl;
7     // Wont compile - value pointed to is const
8     *x = 20;
9     std::cout << "End of function, x = " << *x << std::endl;
10    std::cout << "Address of x at end of function = " << x << std::
        endl;
11 }
12
13 void foo2(int *const x)
14 {
15     std::cout << "Address of x in function = " << x << std::endl;
16     std::cout << "Start of function, x = " << *x << std::endl;
17     // Will compile - pointer is const, not value pointed to
18     *x = 20;
19     std::cout << "End of function, x = " << *x << std::endl;
20     std::cout << "Address of x at end of function = " << x << std::
        endl;
21 }
22
23 void foo3(int *const x)
24 {
25     std::cout << "Address of x in function = " << x << std::endl;
26     std::cout << "Start of function, x = " << *x << std::endl;

```

```

27 // Won't compile - trying to change address pointed to
28 x = nullptr;
29 std::cout << "End of function, x = " << *x << std::endl;
30 std::cout << "Address of x at end of function = " << x << std::
    endl;
31 }
32
33 int main(int argc, char **argv)
34 {
35     int x = 10;
36     std::cout << "Starting address of x = " << &x << std::endl;
37     std::cout << "Before function call, x = " << x << std::endl;
38     // Have to pass the pointer (or address of) x to the function
39     foo2(&x);
40     std::cout << "After function call, x = " << x << std::endl;
41     std::cout << "End address of x = " << &x << std::endl;
42
43     return 0;
44 }

```

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
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:

```
1 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. We will also pass filenames by `const` reference, as the functions don't need to modify the filename.

```
1 #include <iostream>
2 #include <fstream>
3 #include <vector>
4 #include <string>
5
6 using namespace std;
7
8 vector<string> readfile(const string &filename)
9 {
10     // Open file - default is text
11     ifstream file(filename);
12     // String to read into
13     string line;
14     // Data to return
15     vector<std::string> data;
16     // Read until end of file
17     while (getline(file, line))
18         data.push_back(line);
19     // Close file
20     file.close();
21     // Return data
22     return data;
23 }
24
25 void writefile(const string &filename, vector<string> &data)
26 {
27     // Open file - default is text
28     ofstream file(filename);
29     // Write each line into file starting at end of vector
30     for (int i = data.size() - 1; i >= 0; --i)
31         file << data[i] << endl;
32     // Close the file
33     file.close();
34 }
35
36 int main(int argc, char **argv)
37 {
38     // Read in file
39     vector<string> data = readfile(string("sorted.txt"));
40     // Print lines read
41     cout << "Lines read = " << data.size() << endl;
42     writefile(string("reversed.txt"), data);
43
44     return 0;
45 }
```



## 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 people's 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.

```

1 #include <cstdlib>
2 #include <cstdio>
3 #include <climits>
4 #include <cfloat>
5
6 int main(int argc, char **argv)
7 {
8     printf("unsigned char is %d bytes, min value %u, max value %u\n",
9           sizeof(unsigned char), 0, UCHAR_MAX);
10    printf("signed char is %d bytes, min value %d, max value %d\n",
11          sizeof(signed char), SCHAR_MIN, SCHAR_MAX);
12    printf("char is %d bytes, min value %d, max value %d\n", sizeof(
13          char), CHAR_MIN, CHAR_MAX);
14    printf("unsigned short is %d bytes, min value %u, max value %d\n",
15          sizeof(unsigned short), 0, USHRT_MAX);
16    printf("short is %d bytes, min value %d, max value %d\n",
17          sizeof(short), SHRT_MIN, SHRT_MAX);
18    printf("unsigned int is %d bytes, min value %u, max value %u\n",
19          sizeof(unsigned int), 0, UINT_MAX);
20    printf("int is %d bytes, min value %d, max value %d\n", sizeof(
21          int), INT_MIN, INT_MAX);
22    printf("unsigned long is %d bytes, min value %lu, max value %lu\n",
23          sizeof(unsigned long), 0, ULONG_MAX);
24    printf("long is %d bytes, min value %ld, max value %ld\n",
25          sizeof(long), LONG_MIN, LONG_MAX);
26    printf("unsigned long long is %d bytes, min value %llu, max
27          value %llu\n", sizeof(unsigned long long), 0, ULLONG_MAX);
28    printf("long long is %d bytes, min value %lld, max value %lld\n",
29          sizeof(long long), LLONG_MIN, LLONG_MAX);
30    printf("float is %d bytes, min value %e, max value %e\n",
31          sizeof(float), FLT_MIN, FLT_MAX);
32    printf("double is %d bytes, min value %e, max value %e\n",
33          sizeof(double), DBL_MIN, DBL_MAX);

```

```
21     printf("long double is %d bytes, min value %e, max value %e\n",
22           sizeof(long double), LDBL_MIN, LDBL_MAX);
23
24     // Pointer sizes
25     printf("char* is %d bytes\n", sizeof(char*));
26     printf("short* is %d bytes\n", sizeof(short*));
27     printf("int* is %d bytes\n", sizeof(int*));
28
29     return 0;
30 }
```

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:

```
1 x = x + 1; // Don't know what x is here
2 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:

```
1 void func()
2 {
3     // Main scope of the function
4     {
5         // Scope A - can see main scope
6         {
7             // Scope B - can see scope A and main scope
8         }
9
10        {
11            // Scope C - can see scope A and main scope.  Scope
12                B no longer valid
13        }
14 }
```

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.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(int argc, char **argv)
6 {
7     // Outermost declaration
8     int x = 10;
9     cout << "Outermost x = " << x << endl;
```

```

10  {
11      // Now in new scope
12      int x = 20;
13      cout << "\tInner x = " << x << endl;
14      {
15          // Now even further in scope
16          int x = 30;
17          cout << "\t\tInner inner x = " << x << endl;
18          {
19              // Let's stop here
20              int x = 40;
21              cout << "\t\t\tInnermost x = " << x << endl;
22              // Now unwind scope
23          }
24          cout << "\t\tInner inner x = " << x << endl;
25      }
26      cout << "\tInner x = " << x << endl;
27  }
28  cout << "Outermost x = " << x << endl;
29
30  return 0;
31 }

```

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:

```

1  #include <iostream>
2
3  int main(int argc, char **argv)
4  {
5      // Declare an int here
6      int i = 10;
7      {
8          // Declare another int here - can access i
9          int j = i * 2;
10         {
11             // Declare another int here - can access i and j
12             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
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.

```

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

```

```

61 // Free the memory
62 delete[] data;
63 // Set to nullptr
64 data = nullptr;
65
66 return 0;
67 }

```

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.



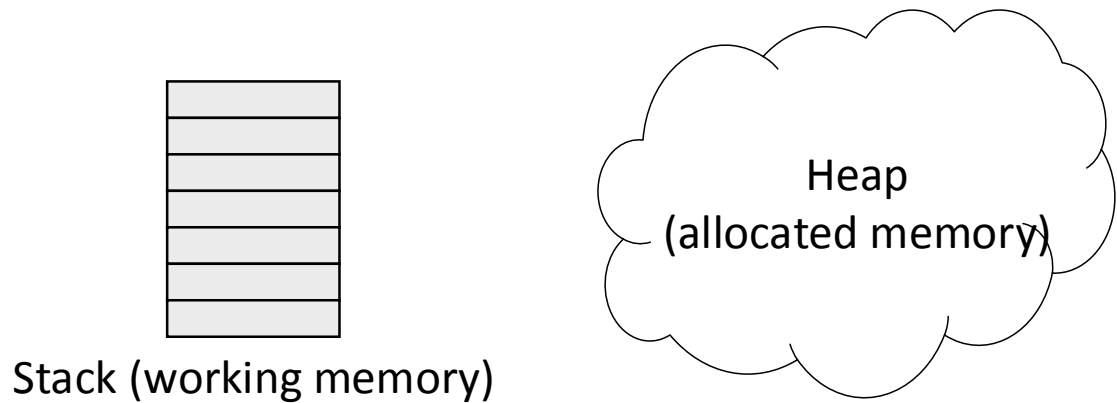


Figure 6.1: Stack and Heap

### 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:

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

**Why do we need to free?**

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

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

**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.

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

```

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

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 than 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:

```

1 #include <cstdlib>
2 #include <stdio>
3
4 // Size of data to allocate
5 const unsigned int SIZE = 1048576;
6
7 int main(int argc, char **argv)
8 {
9     // Try and create data on the stack
10    // This is 1MB - common stack size.
11    // Other values take us over the stack size
12    char data[SIZE];
13    for (int i = 0; i < SIZE; ++i)
14        printf("%d\n", data[i]);
15
16    return 0;
17 }
```

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.

```

1 #include <cstdlib>
2 #include <stdio>
3
4 // 1 megabyte in bytes
5 const unsigned int MB = 1024 * 1024;
6 // 1 gigabyte in bytes
```

```
7 const unsigned int GB = 1024 * MB;
8
9 int main(int argc, char **argv)
10 {
11     // Allocate 1 GB of data
12     char *data = (char*)malloc(GB);
13
14     // Need to "use" the data before actual allocation occurs
15     for (int i = 0; i < 100; ++i)
16         printf("%d\n", data[i]);
17
18     // Free the memory
19     free(data);
20
21     return 0;
22 }
```

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.

```

1 #include <cstdlib>
2 #include <stdio>
3
4 void foo(int *value)
5 {
6     // Allocate the value. This changes the memory pointed to
7     value = (int*)malloc(sizeof(int));
8     *value = 5;
9     // Print address and value
10    printf("In first foo, address - %p, value - %d\n", value, *
        value);
11    // When we return, we lose the memory address
12    // We have a leak!!!
13 }
14
15 void foo(int **value)
16 {
17     // Allocate the value. This changes the memory pointed to
18     *value = (int*)malloc(sizeof(int));
19     **value = 5;
20     // Print address and value - have to dereference twice
21    printf("In second foo, address - %p, value - %d\n", *value, **
        value);
22    // When we return, we retain the pointed to memory address
23 }
24
25 int main(int argc, char **argv)
26 {
27     // Declare value. Use calloc to set as 0
28     int *value = (int*)calloc(1, sizeof(int));
29     // Print address and value
30    printf("Initially, address - %p, value - %d\n", value, *value);
31    // Call foo
32    foo(value);
33    // Print address and value
34    printf("After first foo, address - %p, value - %d\n", value, *
        value);
35    // Call foo with pointer to pointer
36    foo(&value);
37    // Print address and value
38    printf("After second foo, address - %p, value - %d\n", value, *
        value);
39    // Free memory
40    free(value);
41
42    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 = " <<
35             data[i] << endl;
36     // Delete the memory
37     delete[] data2;
38
39     return 0;
40 }

```

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:

```
1 void *mem = malloc(1024);
2 // Do something with the memory
3 free(mem);
4 // Set pointer to null (0)
5 mem = 0;
```

In C++ we would do something like this:

```
1 int *x = new int(5);
2 // Do something with x
3 delete x;
4 x = 0;
```

In modern C++ (C++11 onwards) we can use the `nullptr` value (which also equals 0):

```
1 int *x = new int(5);
2 // Do something with x
3 delete x;
4 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:

```

1 #include <iostream>
2 #include <memory>
3
4 using namespace std;
5
6 void foo(shared_ptr<int> value)
7 {
8     // Print out number of accessors to the shared object - should
9     // be 2
10    cout << "In foo, count on shared_ptr = " << value.use_count()
11    << endl;
12    // shared_ptr behaves like a pointer - we can assign to it by
13    // dereferencing
14    *value = 5;
15 }
16
17 int main(int argc, char **argv)
18 {
19     // We can create a shared_ptr by wrapping it around an
20     // allocated object
21     shared_ptr<int> value = shared_ptr<int>(new int(10));
22     cout << "Initial address = " << value << ", value = " << *value
23     << endl;
24     // We can also create a shared_ptr by using the make_shared
25     // function
26     value = make_shared<int>(70);
27     cout << "After make shared, address = " << value << ", value = "
28     << *value << endl;
29     // Print out number of accessors to the shared object
30     cout << "Before foo, count on shared_ptr = " << value.use_count
31     () << endl;
32     // Let's call foo
33     foo(value);
34     cout << "After foo, address = " << value << ", value = " << *
35     value << endl;
36     // Print out number of accessors to the shared object
37     cout << "After foo, count on shared_ptr = " << value.use_count
38     () << endl;
39     // Let's set value to nullptr - this will clear the memory
40     value = nullptr;
41     // Print out number of accessors to the shared object. Should
42     // be 0
43     cout << "After nullptr, count on shared_ptr = " << value.
44     use_count() << endl;
45
46     return 0;
47 }

```

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 cannot. 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.

**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 copy of the

pointer, which is then lost when the function returns. 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() <<
10     endl;
11     // Call make_shared again - we lose this copy
12     value = make_shared<int>(25);
13     // Output number of accessors to value - should be 1
14     cout << "In foo, after make_shared, accessors to value = " <<
15     value.use_count() << endl;
16     // No memory leaks! shared_ptr will clean itself up
17 }
18
19 // Note we now have a reference to the shared_ptr
20 void foo2(shared_ptr<int> &value)
21 {
22     // Output number of accessors to value - should be 1
23     cout << "In foo2, accessors to value = " << value.use_count()
24     << endl;
25     // Call make_shared again - we retain this copy
26     value = make_shared<int>(100);
27     // Output number of accessors to value - should still be 2
28     cout << "In foo2, after make_shared, accessors to value = " <<
29     value.use_count() << endl;
30 }
31
32 int main(int argc, char **argv)
33 {
34     // Create a new shared_ptr
35     shared_ptr<int> value = make_shared<int>(5);
36     // Print data
37     cout << "Address = " << value << ", value = " << *value << ",
38     usage = " << value.use_count() << endl;
39     // Call foo
40     foo(value);
41     // Print data
42     cout << "Address = " << value << ", value = " << *value << ",
43     usage = " << value.use_count() << endl;
44     // Call foo2
45     foo2(value);
46     // Print data
47     cout << "Address = " << value << ", value = " << *value << ",
48     usage = " << value.use_count() << endl;
49
50     return 0;
51 }
```

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 Examining 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:

```
1 #include <iostream>
2 #include <memory>
3
4 using namespace std;
5
6 const int SIZE = 65536;
7
8 void build_array(shared_ptr<int> data)
9 {
10     for (int i = 0; i < SIZE; ++i)
11         // Have to dereference the pointer, then access the values
12         data.get()[i] = i;
13 }
14
15 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],
27         default_delete<int[]>());
28     // Build the array
29     build_array(data);
30     // Print the array
31     print_array(data);
32     // Set data to nullptr - will cause data to be deleted
33     data = nullptr;
34
35     return 0;
36 }
```

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.

```

1 #include <iostream>
2 #include <memory>
3
4 using namespace std;
5
6 void foo(unique_ptr<int> value)
7 {
8     // Let's set the value to 500. Can treat just as a pointer
9     *value = 500;
10    // We don't return unique_ptr. We have lost *ALL* the data
11 }
12
13 unique_ptr<int> foo2(unique_ptr<int> value)
14 {
15     // This time we will move the answer back
16     *value = 500;
17     return value;
18 }
19
20 void foo3(unique_ptr<int> &value)
21 {
22     // Using a reference solves the problem
23     *value = 1000;
24 }
25
26 int main(int argc, char **argv)
27 {
28     // Create a unique_ptr value
29     unique_ptr<int> value(new int(50));
30     // Print data
31     cout << "Address = " << value.get() << ", value = " << *value
        << endl;
32     // Call foo. We have to "move" the pointer into foo
33     foo(move(value));
34     // Print data. Cannot print value. No longer valid
35     cout << "After foo, address = " << value.get() << endl;
36
37     // 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`.

```

1  #include <iostream>
2  #include <memory>
3
4  using namespace std;
5
6  const int SIZE = 65536;
7
8  void build_array(unique_ptr<int[]> &data)
9  {
10     for (int i = 0; i < SIZE; ++i)
11         // Have to dereference the pointer, then access the values
12         data.get()[i] = i;
13 }
14
15 void print_array(unique_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     // unique_ptr can have arrays
25     unique_ptr<int[]> data(new int[SIZE]);
26     // Build the array - using pass by reference on the unique_ptr
27     build_array(data);
28     // Print the array
29     print_array(data);
30     // Set data to nullptr - will delete the array
31     data = nullptr;
32
33     return 0;
34 }

```

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.

```

1 #include <iostream>
2 #include <memory>
3
4 using namespace std;
5
6 int main(int argc, char **argv)
7 {
8     // Use the auto keyword - compiler determines the type
9     auto data = make_shared<int>(5);
10    // ... data is a shared_ptr<int>. Compiler fills in the type
11    return 0;
12 }

```

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:

```
1 // Type is shared_ptr<int>
2 auto x = make_shared<int>(5);
3 // Now just try and set x to 5 (so just an int)
4 x = 5;
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:

```
1 float x = 12.5f;
2 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:

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 int main(int argc, char **argv)
7 {
8     // 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.

```

1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  int main(int argc, char **argv)
7  {
8      // Create a string
9      string str = "Hello World!";
10     cout << "String = " << str << endl;
11     // Get the raw C string
12     const char *const_raw = str.c_str();
13     // Can't change any values in raw - const
14     // This won't compile
15     // const_raw[4] = 'p';
16     // Now let's live dangerously - use const cast
17     char *raw = const_cast<char*>(const_raw);
18     // Change the value of raw now

```

```
19 raw[4] = 'p';
20 // Print out the string
21 cout << "String = " << str << endl;
22
23 return 0;
24 }
```

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.

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 int main(int argc, char **argv)
7 {
8     // Create a string
9     string str1 = "Hello";
10    string str2 = "World";
11
12    // Check if the types are equal
13    if (typeid(str1) == typeid(str2))
14        cout << "Types are equal" << endl;
15    else
16        cout << "Types are not equal" << endl;
17
18    // Create a float
19    float x = 52.0f;
```



```

20
21 // Print out details
22 cout << "x is a " << typeid(x).name() << " with value " << x <<
    endl;
23
24 return 0;
25 }

```

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.

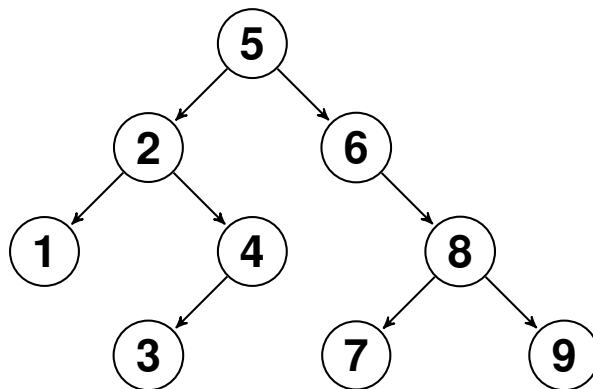


Figure 6.2: Binary Tree Example

**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

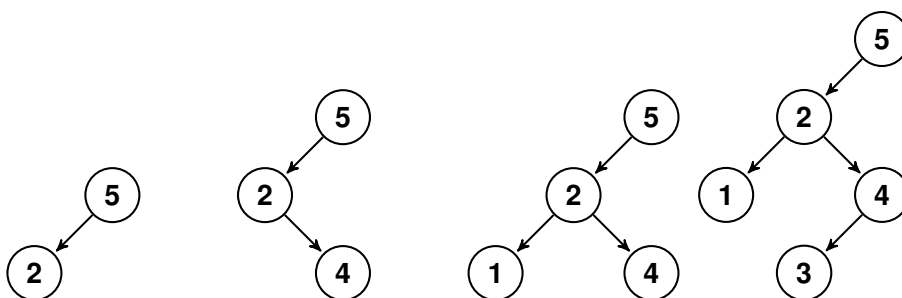


Figure 6.4: Building the Left Branch

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.

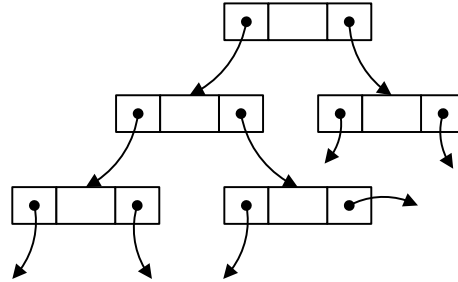


Figure 6.5: Data Structure of a Binary Tree

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:

```

1 struct node
2 {
3     // Data stored in this node of the tree
4     int data;
5     // The left branch of the tree
6     node *left;
7     // The right branch of the tree
8     node *right;
9 };

```

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, ...

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.

```
1 #include <iostream>  
2  
3 using namespace std;  
4  
5 // The node of a tree  
6 struct node  
7 {  
8     // Data stored in this node of the tree  
9     int data;  
10    // The left branch of the tree  
11    node *left;  
12    // The right branch of the tree  
13    node *right;  
14 };  
15  
16 // Inserts a value into the tree - notice **  
17 void insert(node **tree, int value)  
18 {  
19     // TODO  
20 }  
21  
22 // Deletes the tree - freeing memory  
23 void delete_tree(node *tree)  
24 {  
25     // TODO  
26 }  
27  
28 // Prints the tree in order  
29 void inorder(node *tree)  
30 {
```

```

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:

```

1 // Inserts a value into the tree - notice **
2 void insert(node **tree, int value)
3 {
4     // Check if nullptr. If so set new node
5     if (*tree == nullptr)
6     {
7         // Create new node
8         *tree = new node;
9         // Set new value
10        (*tree)->data = value;
11        // Set branches to nullptr
12        (*tree)->left = nullptr;
13        (*tree)->right = nullptr;
14    }
15 }

```

---

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
    Delete tree

```

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

## 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 its 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.

### 7.1 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*.

## 7.2 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.

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 // Student struct in C++
7 // Note the use of string now rather than char*
8 struct student
9 {
10     unsigned int matric;
11     string name;
12     string address;
13 };
14
15 // Prints student information
16 // Note we don't have to use 'struct student' now
17 // Also pass by reference
18 void print_student(student &s)
19 {
20     cout << "matric no: " << s.matric << endl;
21     cout << "name: " << s.name << endl;
22     cout << "address: " << s.address << endl;
23 }
24
25 int main(int argc, char **argv)
26 {
27     // Output sizeof student
28     cout << "Size of student struct is " << sizeof(student) << "
29         bytes" << endl;
30     // Create a student using brackets construction
31     student s =
32     {
33         42001290,
34         "Kevin Chalmers",
35         "School of Computing",
36     };
37     // Print student data
38     print_student(s);
39
40     return 0;
```

41 | }

Listing 7.1: `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.

```
1 Size of student struct is 52 bytes
2 matric no: 42001290
3 name: Kevin Chalmers
4 address: School of Computing
```

Listing 7.2: 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).

## 7.3 Data Sizes and Memory Representation of struct

We discussed the idea of memory representation of `structs` 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 too 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 7.1

Each box represents one of our data types. We have the following:

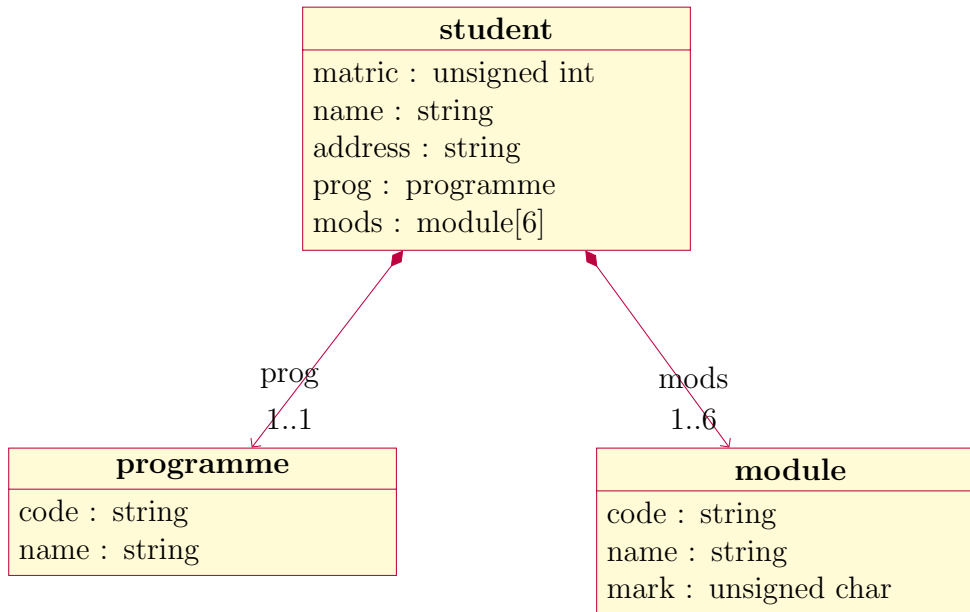


Figure 7.1: Class Diagram for Data Sizes Application

**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.

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 // Represents a programme of study
7 struct programme
8 {
9     string code;
10    string name;
11 };
12
```

```

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 7.3: 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]

### 7.3.1 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 7.4: Output from C++ Data Size Application

## 7.4 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.

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 // Define student as a class
7 class student
8 {
9     // We will come back to public soon
10 public:
11     unsigned int matric;
12     string name;
13     string address;
14 };
15
16 // Print student details
17 void print_student(student &s)
18 {
19     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) << "
28         bytes" << endl;
29     // Create a student using brackets construction
30     student s =
31     {
32         42001290,
33         "Kevin Chalmers",
34         "School of Computing",
35     };
36     // Print student data
37     print_student(s);
38
39     return 0;
40 }

```

Listing 7.5: 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 7.6: Output from class Application

## 7.5 Data Size of a class

Let us now return to our student record application that we implemented using `structs` 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 7.1).

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 // Represents a programme of study
7 class programme
8 {
9 public:
10     string code;
11     string name;
12 };
13
14 // Represents a module studied
15 class module
16 {
17 public:
18     string code;
19     string name;
20     unsigned char mark;
21 };
22
23 // Represents a programme
24 class student
25 {
26 public:
27     unsigned int matric;
28     string name;
29     string address;
30     programme prog;
31     module mods[6];
32 };
33
34 int main(int argc, char **argv)
35 {
36     // Print the size of programme
37     cout << "Size of programme struct: " << sizeof(programme) << "
38         bytes" << endl;
39     // Print the size of module
40     cout << "Size of module struct: " << sizeof(module) << " bytes"
41         << endl;
42     // Print the size of student
43     cout << "Size of student struct: " << sizeof(student) << " bytes"
44         << endl;
45
46     // ***** Add code to prompt the user for details and calculate
47         year average *****
48
49     return 0;
50 }
```

Listing 7.7: Data Size of a Class

### 7.5.1 Exercise

Complete the application as you did previously with the **struct** version of the application. The output is the same as before.



## 7.6 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.

```

1 #include <iostream>
2
3 using namespace std;
4
5 // Struct with public and private members
6 struct my_struct
7 {
8     // Default access of struct is public
9     unsigned int x = 10;
10 private:
11     // This value is not accessible outside my_struct
12     unsigned int y = 20;
13 };
14
15 class my_class
16 {
17     // Default access of class is private
18     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 7.8: 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 memb
er 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 memb
er 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 memb
er 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 7.9: 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.

## 7.7 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 its 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:

```
1 class my_class
2 {
3 private:
4     int x;
5     int y;
6 public:
7     int add()
8     {
9         return x + y;
10    }
11};
```

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:

```
1 class my_class
2 {
3 private:
4     int x;
5     int y;
6 public:
7     int add();
8 };
9
10 // Other code, maybe even a separate file
11
12 int my_class::add()
13 {
14     return x + y;
15 }
```

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.

```
1 #include <iostream>
2
3 using namespace std;
4
5 struct my_struct
6 {
7     // Publicly accessible
8     unsigned int x = 10;
9 private:
```

```

10 // Privately accessible
11 unsigned int y = 20;
12
13 public:
14 // Public methods
15 unsigned int get_y()
16 {
17     return y;
18 }
19
20 void set_y(unsigned int value)
21 {
22     y = value;
23 }
24 };
25
26 class my_class
27 {
28 // Privately accessible
29 unsigned int x = 10;
30 public:
31 // Publicly accessible
32 unsigned int y = 20;
33
34 // Public methods
35 unsigned int get_x()
36 {
37     return x;
38 }
39
40 void set_x(unsigned int value)
41 {
42     x = value;
43 }
44 };
45
46 int main(int argc, char **argv)
47 {
48 // Declare objects
49 my_struct a;
50 my_class b;
51
52 // Can access a.x directly
53 cout << "a.x = " << a.x << endl;
54 // Use get_y on a to access a.y
55 cout << "a.y = " << a.get_y() << endl;
56 // Use get_x on b to access b.x
57 cout << "b.x = " << b.get_x() << endl;
58 // This line will compile
59 cout << "b.y = " << b.y << endl;
60
61 // Try and set the values
62 a.x = 20;
63 a.set_y(40);
64 b.set_x(30);
65 b.y = 60;
66
67 // Can access a.x directly
68 cout << "a.x = " << a.x << endl;
69 // Use get_y on a to access a.y

```

```
70 cout << "a.y = " << a.get_y() << endl;
71 // Use get_x on b to access b.x
72 cout << "b.x = " << b.get_x() << endl;
73 // This line will compile
74 cout << "b.y = " << b.y << endl;
75
76 return 0;
77 }
```

Listing 7.10: 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 7.11: Output from Methods Application

## 7.8 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++:

```
1 class my_class
2 {
3 private:
4     int x;
5     int y;
6 public:
7     my_class()
8     {
9         x = 0;
10        y = 0;
11    }
12};
```

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:

```
1 class my_class
2 {
3 private:
4     int x;
5     int y;
6 public:
7     my_class()
8     {
9         x = 0;
10        y = 0;
11    }
12    my_class(int a, int b)
13    {
14        x = a;
15        y = b;
16    }
17 };
```

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.

```
1 #include <iostream>
2
3 using namespace std;
4
5 struct my_struct
6 {
7     unsigned int x;
8     unsigned int y;
9
10    my_struct(unsigned int a, unsigned int b)
11    {
12        x = a;
13        y = b;
14    }
15 };
16
17 class my_class
18 {
19 public:
20     unsigned int x;
21     unsigned int y;
22 }
```

```
23 my_class(unsigned int a, unsigned int b)
24 {
25     x = a;
26     y = b;
27 }
28 };
29
30 int main(int argc, char **argv)
31 {
32     // Create objects
33     my_struct a(10, 20);
34     my_class b(10, 20);
35
36     // Print details
37     cout << "a.x = " << a.x << endl;
38     cout << "a.y = " << a.y << endl;
39     cout << "b.x = " << b.x << endl;
40     cout << "b.y = " << b.y << endl;
41
42     // Reallocate a and b
43     a = my_struct(20, 40);
44     b = my_class(30, 60);
45
46     // Print details
47     cout << "a.x = " << a.x << endl;
48     cout << "a.y = " << a.y << endl;
49     cout << "b.x = " << b.x << endl;
50     cout << "b.y = " << b.y << endl;
51
52     return 0;
53 }
```

Listing 7.12: 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.

## 7.9 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.

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 // Defines a programme of study
7 struct programme
8 {
9     string code;
10    string name;
11
12    // Default constructor
13    programme()
14    {
15        cout << "Called programme default constructor" << endl;
16    }
17
18    // Constructor with parameters
19    programme(const string &code, const string &name)
20    {
21        cout << "Called programme parametrised constructor" << endl;
22        this->code = code;
23        this->name = name;
24    }
25 };
26
27 // Defines a module
28 struct module
29 {
30     string code;
31     string name;
32     unsigned char mark;
33
34     // Default constructor
35     module()
36     {
37         cout << "Called module default constructor" << endl;
38     }
39
40     // Constructor with parameters
41     module(const string &code, const string &name, unsigned int mark)
42     {
43         cout << "Called module parametrised constructor" << endl;
44         this->code = code;
45         this->name = name;
46         this->mark = mark;
```

```

47     }
48 };
49
50 // Defines a student
51 struct student
52 {
53     unsigned int matric;
54     string name;
55     string address;
56     programme prog;
57     module mods[6];
58
59     // Default constructor
60     student()
61     {
62         cout << "Called student default constructor" << endl;
63     }
64 };
65
66 int main(int argc, char **argv)
67 {
68     cout << "Creating a student object" << endl;
69     student s;
70
71     // Set some student values
72     cout << "Setting student details" << endl;
73     s.matric = 42001290;
74     s.name = "Kevin Chalmers";
75     s.address = "School of Computing";
76
77     // Set the programme
78     cout << "Setting programme" << endl;
79     s.prog = programme("56119BH", "BEng (Hons) Software Engineering")
80         ;
81
82     // Set the modules
83     cout << "Setting module 0" << endl;
84     s.mods[0] = module("SET07109", "Programming Fundamentals", 85);
85
86     // **** Add other modules ****
87
88     // **** Calculate average mark ****
89
90     // **** Print average mark ****
91
92     return 0;
93 }

```

Listing 7.13: 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 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 7.14: 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.

## 7.9.1 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.

### 7.9.2 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.

```

1 class A
2 {
3 private:
4     string str;
5 public:
6     A(const string &s)
7       : str(s) // Calls the string constructor
8     {
9     }
10 };
11
12 class B
13 {
14 private:
15     string str;
16     A a;
17 public:
18     B(const string &s)
19       : str(s), a("Hello") // Calls string and A constructor
20     {
21     }
22 };

```

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).

## 7.10 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:

```
1 class my_class
2 {
3     // other declarations
4 public:
5     ~my_class()
6     {
7         // clean up resources
8     }
9 };
```

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.

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 // Defines a programme of study
7 struct programme
8 {
9     string code;
10    string name;
11
12    // Default constructor
13    programme()
14    {
15        cout << "Called programme default constructor" << endl;
16    }
17
18    // Constructor with parameters
19    programme(const string &code, const string &name)
20    {
21        cout << "Called programme parametrised constructor" << endl;
22        this->code = code;
23        this->name = name;
24    }
25
26    // Destructor for programme
27    ~programme()
28    {
29        cout << "Called programme destructor" << endl;
30    }
31 };
32
33 // Defines a module
34 struct module
```

```

35 {
36     string code;
37     string name;
38     unsigned char mark;
39
40     // Default constructor
41     module()
42     {
43         cout << "Called module default constructor" << endl;
44     }
45
46     // Constructor with parameters
47     module(const string &code, const string &name, unsigned int mark)
48     {
49         cout << "Called module parametrised constructor" << endl;
50         this->code = code;
51         this->name = name;
52         this->mark = mark;
53     }
54
55     // Module destructor
56     ~module()
57     {
58         cout << "Called module destructor" << endl;
59     }
60 };
61
62 // Defines a student
63 struct student
64 {
65     unsigned int matric;
66     string name;
67     string address;
68     programme prog;
69     module mods[6];
70
71     // Default constructor
72     student()
73     {
74         cout << "Called student default constructor" << endl;
75     }
76
77     // Student destructor
78     ~student()
79     {
80         cout << "Called student destructor" << endl;
81     }
82 };
83
84 int main(int argc, char **argv)
85 {
86     cout << "Creating a student object" << endl;
87     student s;
88
89     // Set some student values
90     cout << "Setting student details" << endl;
91     s.matric = 42001290;
92     s.name = "Kevin Chalmers";
93     s.address = "School of Computing";
94

```

```

95 // Set the programme
96 cout << "Setting programme" << endl;
97 s.prog = programme("56119BH", "BEng (Hons) Software Engineering")
98 ;
99 // Set the modules
100 cout << "Setting module 0" << endl;
101 s.mods[0] = module("SET07109", "Programming Fundamentals", 85);
102
103 // **** Add other modules ****
104
105 // **** Calculate average mark ****
106
107 // **** Print average mark ****
108
109 return 0;
110 }

```

Listing 7.15: 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 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 module destructor
Called programme destructor

```

Listing 7.16: 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 allocating your objects only once.

### 7.10.1 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.

### 7.10.2 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.

## 7.11 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 been 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.

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 // Defines a programme of study
7 struct programme
8 {
9     string code;
10    string name;
11
12    // Default constructor
13    programme()
14    {
15        cout << "Called programme default constructor" << endl;
16    }
17
18    // Constructor with parameters
19    programme(const string &code, const string &name)
20    {
21        cout << "Called programme parametrised constructor" << endl;
22        this->code = code;
23        this->name = name;
24    }
25
26    // Destructor for programme
27    ~programme()
28    {
29        cout << "Called programme destructor" << endl;
30    }
31 };
32
33 // Defines a module
34 struct module
35 {
36     string code;
37     string name;
38     unsigned char mark;
39
40    // Default constructor
41    module()
42    {
43        cout << "Called module default constructor" << endl;
44    }
45 }
```

```

46 // Constructor with parameters
47 module(const string &code, const string &name, unsigned int mark)
48 {
49     cout << "Called module parametrised constructor" << endl;
50     this->code = code;
51     this->name = name;
52     this->mark = mark;
53 }
54
55 // Module destructor
56 ~module()
57 {
58     cout << "Called module destructor" << endl;
59 }
60 };
61
62 // Defines a student
63 struct student
64 {
65     unsigned int matric;
66     string name;
67     string address;
68     programme prog;
69     module mods[6];
70
71     // Default constructor
72     student()
73     {
74         cout << "Called student default constructor" << endl;
75     }
76
77     // Programme destructor
78     ~student()
79     {
80         cout << "Called student destructor" << endl;
81     }
82 };
83
84 int main(int argc, char **argv)
85 {
86     cout << "Creating a student object on the heap" << endl;
87     student *s = new student();
88
89     // Set some student values
90     cout << "Setting student details" << endl;
91     s->matric = 42001290;
92     s->name = "Kevin Chalmers";
93     s->address = "School of Computing";
94
95     // Set the programme
96     cout << "Setting programme" << endl;
97     s->prog = programme("56119BH", "BEng (Hons) Software Engineering"
98         );
99
100    // Set the modules
101    cout << "Setting module 0" << endl;
102    s->mods[0] = module("SET07109", "Programming Fundamentals", 85);
103
104    // **** 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 7.17: 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.

### 7.11.1 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`.

## 7.12 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.

### 7.12.1 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.

```
1 student *s = new student();
2 // ... do some other work
3 student a = *s;
4 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:

```

1 student *s = new student();
2 // ... do some other work
3 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.

```

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     // Need to dereference the pointer
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 7.18: Accessing Members by Dereferencing a Pointer

You should be able to predict the output from this application by now.

### 7.12.2 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 7.19: 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.

## 7.13 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 7.2 illustrates this.

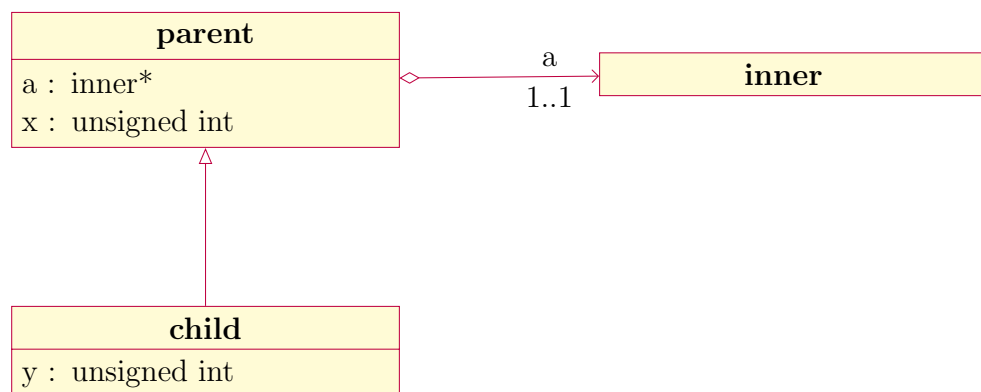


Figure 7.2: Class Diagram Showing Inheritance

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:

```
1 class animal
2 {
3     // .. animal declaration
4 };
5
6 class mammal : public animal
7 {
8     // ... mammal declaration
9 };
10
11 class dog : public mammal
12 {
13     // ... dog declaration
14 };
```

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.

```
1 #include <iostream>
2
3 using namespace std;
4
5 // Inner class
6 class inner
7 {
8 public:
9     inner()
10    {
11        cout << "Called inner constructor" << endl;
12    }
13    ~inner()
```



```

14     {
15         cout << "Called inner destructor" << endl;
16     }
17 };
18
19 class parent
20 {
21 private:
22     // Parent contains a pointer to an inner object
23     inner *a;
24 public:
25     // Parent contains an unsigned int value
26     unsigned int x;
27
28     parent()
29     {
30         cout << "Called parent constructor" << endl;
31         // Calls default constructor
32         a = new inner;
33     }
34     ~parent()
35     {
36         cout << "Called parent destructor" << endl;
37         delete a;
38     }
39 };
40
41 class child : public parent
42 {
43 public:
44     // Child adds an unsigned int value
45     unsigned int y;
46
47     child()
48     {
49         cout << "Called child constructor" << endl;
50     }
51     ~child()
52     {
53         cout << "Called child destructor" << endl;
54     }
55 };
56
57 int main(int argc, char **argv)
58 {
59     // Print out size of inner
60     cout << "sizeof(inner) = " << sizeof(inner) << " bytes" << endl;
61     // Print out size of parent
62     cout << "sizeof(parent) = " << sizeof(parent) << " bytes" << endl;
63     ;
64     // Print out size of child
65     cout << "sizeof(child) = " << sizeof(child) << " bytes" << endl;
66
67     // Create a child object on the heap
68     // Calls default constructor
69     child *c = new child;
70
71     // Delete the child object
72     delete c;

```

```
73 |   return 0;  
74 | }
```

Listing 7.20: 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 classes 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 7.21: 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.

### 7.13.1 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.

## 7.14 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
<b>public</b>	yes	yes	yes
<b>private</b>	yes	no	no
<b>protected</b>	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.

```
1 #include <iostream>
2
3 using namespace std;
4
5 class parent
6 {
7 private:
8     // This is only accessible in parent
9     unsigned int x;
10 protected:
11     // This is accessible in parent and its children
12     unsigned int y;
13 public:
14     // This is accessible in parent, children and externally
15     unsigned int z;
16
17     void print_parent()
18     {
19         cout << "parent = {" << x << "," << y << "," << "," << "}" <<
20             endl;
21     }
22 };
23
24 class child : public parent
25 {
26 private:
27     // This is only accessible in child
```

```

27     unsigned int a;
28 protected:
29     // This is accessible in child and its children
30     unsigned int b;
31 public:
32     // This is accessible in child, children and externally
33     unsigned int c;
34
35     void print_child()
36     {
37         // Parent details - x not accessible. Won't compile
38         cout << "parent = {" << x << "," << y << "," << z << "}" <<
            endl;
39         // Child details
40         cout << "child = {" << a << "," << b << "," << c << "}" << endl
            ;
41     }
42 };
43
44 int main(int argc, char **argv)
45 {
46     // Print sizeof parent
47     cout << "sizeof(parent) = " << sizeof(parent) << endl;
48     // Print sizeof child
49     cout << "sizeof(child) = " << sizeof(child) << endl;
50
51     // Create a child
52     child ch;
53
54     // Access values of ch
55     // Won't compile - not accessible
56     cout << "ch.x = " << ch.x << endl;
57     // Won't compile - not accessible
58     cout << "ch.y = " << ch.y << endl;
59     // Will compile - z publicly accessible
60     cout << "ch.z = " << ch.z << endl;
61     // Won't compile - not accessible
62     cout << "ch.a = " << ch.a << endl;
63     // Won't compile - not accessible
64     cout << "ch.b = " << ch.b << endl;
65     // Will compile - c publicly accessible
66     cout << "ch.c = " << ch.c << endl;
67
68     // Call print parent on ch
69     ch.print_parent();
70     // Call print child on ch
71     ch.print_child();
72
73     return 0;
74 }

```

Listing 7.22: 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 7.23: Compiler Output from `protected` Application

### 7.14.1 Exercise

Fix this application by introducing a method that allows accessing of the values in the class. These methods will need to be `public`.

## 7.15 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:

```
1 class my_class
2 {
3 private:
4     // x is a static value - belongs to the class
5     static int x;
6 public:
7     // get_x is a static method - belongs to the class
8     static int get_x() { return x; }
9 };
```

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:

```
1 #include <iostream>
2
3 using namespace std;
4
5 class my_class
6 {
7 private:
8     // This value is stored in the class - not an object
9     static unsigned int x;
10 public:
11     // This method is called via the class
12     static unsigned int get_x()
13     {
14         return x;
15     }
16     // This method is called via the class
17     static void set_x(unsigned int value)
18     {
19         x = value;
20     }
21 };
22
23 // We have to initialise my_class::x
24 unsigned int my_class::x = 10;
25
26 int main(int argc, char **argv)
27 {
28     // Print the sizeof my_class - x is not counted
29     cout << "sizeof(my_class) = " << sizeof(my_class) << endl;
30
31     // Print x
```

```

32 cout << "my_class::x = " << my_class::get_x() << endl;
33 // Set x
34 my_class::set_x(20);
35 // Print x
36 cout << "my_class::x = " << my_class::get_x() << endl;
37
38 return 0;
39 }

```

Listing 7.24: 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 7.25: Output from static Application

## 7.16 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:

```

1 #include <iostream>
2
3 using namespace std;
4
5 // Inner class
6 class inner
7 {
8 public:
9     inner()
10    {
11        cout << "Called inner constructor" << endl;
12    }
13    virtual ~inner()
14    {
15        cout << "Called inner destructor" << endl;
16    }
17 };
18
19 class parent
20 {
21 private:
22     // Parent contains a pointer to an inner object
23     inner *a;
24 public:

```



```

25 // Parent contains an unsigned int value
26 unsigned int x;
27
28 parent()
29 {
30     cout << "Called parent constructor" << endl;
31     // Calls default constructor
32     a = new inner;
33 }
34 virtual ~parent()
35 {
36     cout << "Called parent destructor" << endl;
37     delete a;
38 }
39 };
40
41 class child : public parent
42 {
43 public:
44     // Child adds an unsigned int value
45     unsigned int y;
46
47     child()
48     {
49         cout << "Called child constructor" << endl;
50     }
51     ~child()
52     {
53         cout << "Called child destructor" << endl;
54     }
55 };
56
57 int main(int argc, char **argv)
58 {
59     // Create a child class. Use parent as type
60     parent *p = new child();
61     // Create a new inner class
62     inner *i = new inner();
63
64     // Convert p to a child using dynamic_cast
65     child *c = dynamic_cast<child*>(p);
66     // Print out memory location
67     cout << "Memory location of p dynamically cast to child: " << c
68         << endl;
69     // Do the same for inner
70     c = dynamic_cast<child*>(i);
71     // Print out memory location
72     cout << "Memory location of i dynamically cast to child: " << c
73         << endl;
74     // Now do the same with reinterpret_cast
75     c = reinterpret_cast<child*>(p);
76     cout << "Memory location of p reinterpreted to child: " << c <<
77         endl;
78     c = reinterpret_cast<child*>(i);
79     cout << "Memory location of i reinterpreted to child: " << c <<
80         endl;
81
82     // Delete objects
83     delete i;

```

```

81     delete p;
82
83     return 0;
84 }

```

Listing 7.26: `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 7.27: 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.

## 7.17 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.

```

1  #include <iostream>
2  #include <fstream>
3  #include <vector>
4
5  using namespace std;
6
7  // The struct we will write out
8  struct employee
9  {
10     unsigned int number;

```

```

11     char name[100];
12     char job_title[100];
13     float salary;
14
15     void print()
16     {
17         cout << number << " " << name << " " << job_title << " " <<
            salary << endl;
18     }
19 };
20
21 void write_employee(employee *emp, const string &filename)
22 {
23     // Open file for writing
24     ofstream output;
25     output.open(filename.c_str(), ios::out | ios::app | ios::binary
26         );
27     // Write employee - note the cast to char*
28     output.write((char*)(emp), sizeof(employee));
29     output.close();
30 }
31 void read_employees(vector<employee> &emp, const string &filename)
32 {
33     // Open file for reading
34     ifstream input;
35     input.open(filename.c_str(), ios::in | ios::binary);
36     // Loop until end of file
37     employee e;
38     while(input.read((char*)&e, sizeof(employee)))
39         emp.push_back(e);
40 }
41
42 int main(int argc, char **argv)
43 {
44     // Loop until number is 0
45     while (true)
46     {
47         employee e;
48         cout << "Employee number: ";
49         cin >> e.number;
50         if (e.number == 0)
51             break;
52         cout << "Employee name: ";
53         cin >> e.name;
54         cout << "Employee job: ";
55         cin >> e.job_title;
56         cout << "Employee wage: ";
57         cin >> e.salary;
58         write_employee(&e, "employees.dat");
59     }
60
61     vector<employee> emp;
62     read_employees(emp, "employees.dat");
63
64     for (auto &e : emp)
65         e.print();
66
67     return 0;
68 }

```

---

Listing 7.28: 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.

## 7.18 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 8

## 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.

### 8.1 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.

```
1 #include <iostream>
2
3 using namespace std;
4
5 class parent
```

```
6 {
7 public:
8     void print()
9     {
10         cout << "Calling parent.print" << endl;
11     }
12 };
13
14 class child : public parent
15 {
16 public:
17     void print()
18     {
19         cout << "Calling child.print" << endl;
20     }
21 };
22
23 int main(int argc, char **argv)
24 {
25     // Create a parent object
26     parent p;
27     // Call print
28     cout << "Calling print on a parent" << endl;
29     p.print();
30
31     // Create a child object
32     child c;
33     // Call print
34     cout << "Calling print on a child" << endl;
35     c.print();
36
37     // Create a child on the heap
38     child *c2 = new child();
39     // Call print on c2
40     cout << "Calling print on a child" << endl;
41     c2->print();
42
43     // Cast child to a parent
44     // Use dynamic_pointer_cast for smart pointers
45     parent *p2 = (parent*)c2;
46     // Call print on p2
47     cout << "Calling print on a child cast to a parent" << endl;
48     p2->print();
49
50     delete c2;
51
52     return 0;
53 }
```

Listing 8.1: 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 8.2: 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`.

## 8.2 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 8.3: 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.

## 8.3 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`.

```
1 #include <iostream>
2
3 using namespace std;
4
5 class parent
6 {
7 public:
8     virtual void print()
9     {
10         cout << "Calling parent.print" << endl;
11     }
12 };
13
14 class child : public parent
15 {
16 public:
17     void print()
18     {
19         cout << "Calling child.print" << endl;
```

```

20 }
21 };
22
23 int main(int argc, char **argv)
24 {
25     // Create a parent object
26     parent p;
27     // Call print
28     p.print();
29
30     // Create a child object
31     child c;
32     // Call print
33     c.print();
34
35     // Create a child on the heap
36     child *c2 = new child();
37     // Call print on c2
38     c2->print();
39
40     // Cast child to a parent
41     parent *p2 = (parent*)c2;
42     // Call print on p2
43     p2->print();
44
45     delete c2;
46
47     return 0;
48 }

```

Listing 8.4: 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 8.5: Output from Correct `virtual` Application

## 8.4 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 [8.1](#). 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.



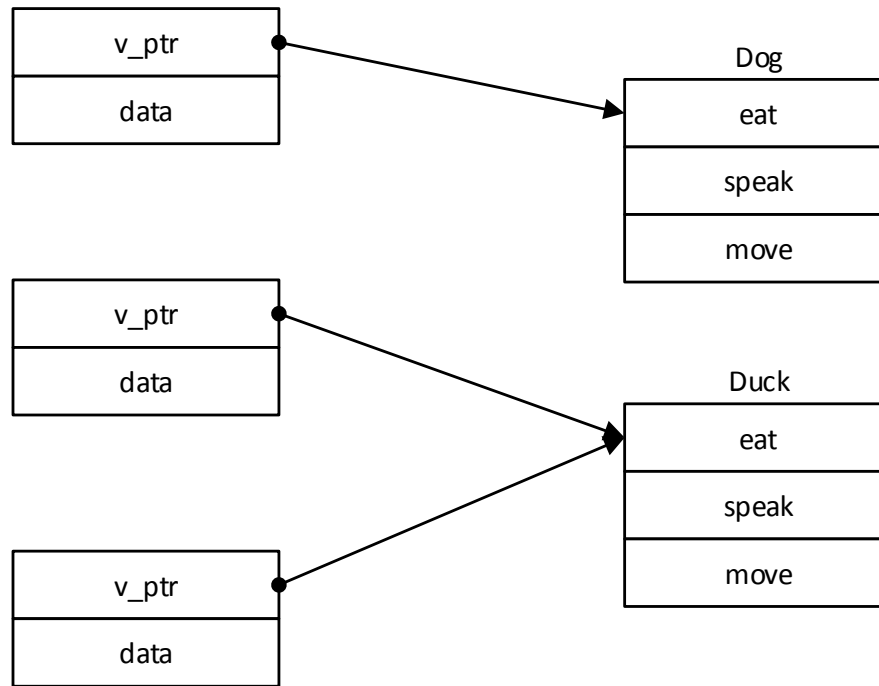


Figure 8.1: Virtual Method Table

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.

## 8.5 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:

```

1 #include <iostream>
2
3 using namespace std;
4
5 class parent
6 {
7 public:
8     // Constructor
9     parent()
10    {
11        cout << "Called parent constructor" << endl;
12    }
13    // Destructor
14    ~parent()

```

```
15 {
16     cout << "Called parent destructor" << endl;
17 }
18 };
19
20 class child : public parent
21 {
22 public:
23     // Constructor
24     child()
25     {
26         cout << "Called child constructor" << endl;
27     }
28     // Destructor
29     ~child()
30     {
31         cout << "Called child destructor" << endl;
32     }
33 };
34
35 int main(int argc, char **argv)
36 {
37     // Create a child on the heap
38     child *a = new child;
39     // Destroy the child
40     delete a;
41
42     // Create a child on the heap
43     // Treat as parent
44     parent *b = new child;
45     // Destroy the child
46     delete b;
47
48     return 0;
49 }
```

Listing 8.6: Destructors with Inheritance

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 8.7: 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.

## 8.6 virtual Destructors

Our test application rewritten with a `virtual` destructor is as follows.

```

1 #include <iostream>
2
3 using namespace std;
4
5 class parent
6 {
7 public:
8     // Constructor
9     parent()
10    {
11        cout << "Called parent constructor" << endl;
12    }
13    // Destructor
14    virtual ~parent()
15    {
16        cout << "Called parent destructor" << endl;
17    }
18 };
19
20 class child : public parent
21 {
22 public:
23     // Constructor
24     child()
25     {
26         cout << "Called child constructor" << endl;
27     }
28     // Destructor
29     ~child()
30     {
31         cout << "Called child destructor" << endl;
32     }
33 };
34
35 int main(int argc, char **argv)
36 {
37     // Create a child on the heap
38     child *a = new child;
39     // Destroy the child
40     delete a;
41
42     // Create a child on the heap
43     // Treat as parent
44     parent *b = new child;
45     // Destroy the child
46     delete b;

```

```
47|
48|     return 0;
49| }
```

Listing 8.8: 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 8.9: Correct Output with virtual Destructors

## 8.7 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**.

### 8.7.1 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:

```
1 class A
2 {
3     virtual void foo();
4     void bar();
5 };
6
7 class B : public A
```

```

8| {
9|     void foo();
10|    void bar();
11| };

```

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.

```

1| class A
2| {
3|     virtual void foo();
4|     void bar();
5| };
6|
7| class B : public A
8| {
9|     void foo() override;
10|    void bar() override;
11| };

```

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.

```

1| #include <iostream>
2|
3| using namespace std;
4|
5| class parent
6| {
7| public:
8|     virtual void my_method()
9|     {
10|         cout << "Called parent.my_method()" << endl;
11|     }
12| };
13|
14| class child : public parent
15| {
16| public:
17|     void my_method() override
18|     {
19|         cout << "Called child.my_method()" << endl;
20|     }
21| };
22|
23| int main(int argc, char **argv)
24| {
25|     // Create a parent object on the heap
26|     parent *p = new parent;
27|     // 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 8.10: 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 8.11: 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 8.2. Here we have an application with three levels of inheritance.

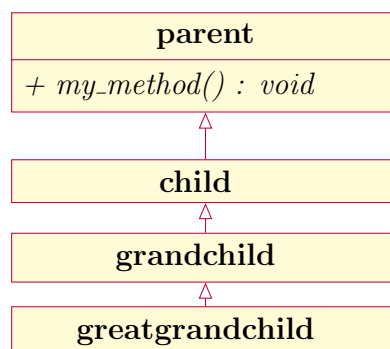


Figure 8.2: Great Grandchild Application Diagram

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.

```

1 #include <iostream>
2
3 using namespace std;
4
5 class parent
6 {
7 public:
8     virtual void my_method()
9     {
10         cout << "Called parent.my_method()" << endl;
11     }
12 };
13
14 class child : public parent
15 {
16 public:
17     void my_method() override
18     {
19         cout << "Called child.my_method()" << endl;
20     }
21 };
22
23 class grandchild : public child
24 {
25     // Not declaring anything
26 };
27
28 class great_grandchild : public grandchild
29 {
30 public:
31     void my_method() override
32     {
33         cout << "Called great_grandchild.my_method()" << endl;
34     }
35 };
36
37 int main(int argc, char **argv)
38 {
39     // Create all object types on the heap
40     parent *p[4];
41     p[0] = new parent;
42     p[1] = new child;
43     p[2] = new grandchild;
44     p[3] = new great_grandchild;
45
46     // Call my_method on each
47     for (int i = 0; i < 4; ++i)
48         p[i]->my_method();
49
50     // delete all the objects
51     for (int i = 0; i < 4; ++i)
52         delete p[i];
53
54     return 0;
55 }

```

Listing 8.12: 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 8.13: 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.

```
1 #include <iostream>
2
3 using namespace std;
4
5 class base
6 {
7 public:
8     virtual void print(unsigned int x)
9     {
10         cout << "Calling base.print with " << x << endl;
11     }
12 };
13
14 class derived1 : public base
15 {
16 public:
17     // Compiling this will cause an error - no override
18     void print(float x) override
19     {
20         cout << "Calling derived1.print with " << x << endl;
21     }
22 };
23
24 class derived2 : public base
25 {
26 public:
27     // Compiling this will not cause an error - hides previous
28     // behaviour
29     void print(float x)
30     {
31         cout << "Calling derived2.print with " << x << endl;
32     }
33 };
34
35 int main(int argc, char **argv)
36 {
37     base *b1 = new derived1;
38     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 8.14: 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 specifier 'override' did not override any base class methods

```

Listing 8.15: 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.

### 8.7.2 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.

```

1  #include <iostream>
2
3  using namespace std;
4
5  class parent
6  {
7  public:
8      virtual void my_method()
9      {
10         cout << "Called parent.my_method()" << endl;
11     }
12 };
13
14 class child : public parent
15 {
16 public:
17     void my_method() override final
18     {
19         cout << "Called child.my_method()" << endl;
20     }
21 };
22
23 class grandchild : public child
24 {
25 public:
26     // This won't compile. Cannot override a final

```

```
27 void my_method() override
28 {
29     cout << "Called grandchild.my_method()" << endl;
30 }
31 };
32
33 int main(int argc, char **argv)
34 {
35     // Create a grandchild object
36     parent *p = new grandchild;
37
38     // Call my_method
39     p->my_method();
40
41     // Delete object
42     delete p;
43 }
```

Listing 8.16: 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' cannot be overridden by 'grandchild::my_method'
final.cpp(17) : see declaration of 'child::my_method'
```

Listing 8.17: Compiler Output from Trying to override `final` Methods

## 8.8 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 cer-

tain 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 8.18: 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 8.19: Compiler Error Trying to Create Abstract Class

### 8.8.1 Exercise

Fix the above application so that it will compile.

## 8.9 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 8.3 provides a diagram of the inheritance hierarchy we just defined. You should try and implement this yourself first before looking at the code below.

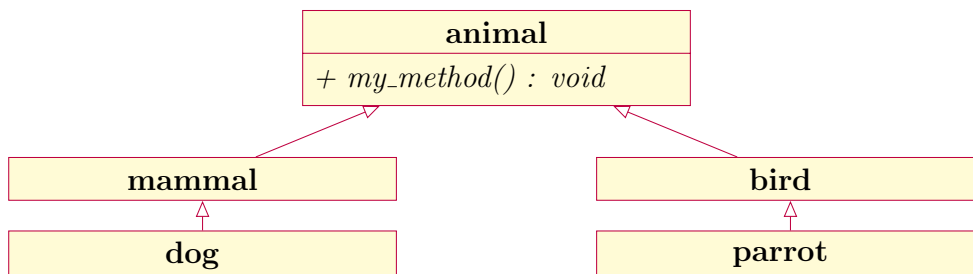


Figure 8.3: Animal Class Diagram

```

1 #include <iostream>
2
3 using namespace std;
4
5 class animal
6 {
7 public:
8     // **** ALWAYS DEFINE THE VIRTUAL DESTRUCTOR ****
9     virtual ~animal()
10    {
11        cout << "Called animal destructor" << endl;
12    }
13
14     virtual void speak() = 0;
15 };
16
17 class bird : public animal

```

```

18 {
19 public:
20     ~bird()
21     {
22         cout << "Called bird destructor" << endl;
23     }
24
25     // Not implementing anything - still an interface
26 };
27
28 class mammal : public animal
29 {
30 public:
31     ~mammal()
32     {
33         cout << "Called mammal destructor" << endl;
34     }
35
36     // Not implementing anything - still an interface
37 };
38
39 class parrot : public bird
40 {
41 public:
42     parrot()
43     {
44         cout << "Called parrot constructor" << endl;
45     }
46
47     ~parrot()
48     {
49         cout << "Called parrot destructor" << endl;
50     }
51
52     // Speak implemented here
53     void speak() override final
54     {
55         cout << "Squak!" << endl;
56     }
57 };
58
59 class dog : public mammal
60 {
61 public:
62     dog()
63     {
64         cout << "Called dog constructor" << endl;
65     }
66
67     ~dog()
68     {
69         cout << "Called dog destructor" << endl;
70     }
71
72     // Speak implemented here
73     void speak() override final
74     {
75         cout << "Bark!" << endl;
76     }
77 };

```

```
78
79 int main(int argc, char **argv)
80 {
81     // Create a parrot and a dog
82     bird *p = new parrot;
83     mammal *d = new dog;
84
85     // Tell the animals to speak
86     p->speak();
87     d->speak();
88
89     // Destroy the animals
90     delete p;
91     delete d;
92
93     return 0;
94 }
```

Listing 8.20: 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 8.21: Output from Interface Application

## 8.10 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 9

## Operator Overloading

We are now going to cover a powerful yet often overlooked capability of object-orientation - *operator overloading*. Operator overloading allows us to define how our classes behave when used with some of the standard operators (e.g. `+`, `-`, `*`, etc.).

So far our journey through this module has been as follows:

1. Understanding the compilation process
2. Understanding data representation
3. Understanding how our high level code is represented on the machine
4. Understanding how the pre-processor works and how we can build libraries of code
5. Understanding what happens when we call a method in regards to the parameters
6. Understanding how we work with memory
7. Debugging applications and approaches to fix bugs
8. Understanding the basics of object-orientation
9. Understanding virtual behaviour and how we can override it

This unit follows on from the work on object-orientation and virtual behaviour to allow us to define behaviour around standard operators. You might think *why do we need this?* Well, you have already been using it. When we worked with a `vector` we were able to access values using indexes (e.g. `vec[4] = 5;`). This is overridden behaviour. Also, the use of `+` to join two `strings` together is overridden behaviour.

So why is this an often neglected area of object-orientation? Because Java doesn't support it. The developers of Java decided this was a complex and unnecessary area. However, when you try and work without object-orientation in some areas (in particular more mathematical ones) you will find out the problems involved.

### 9.1 Operator Overloading

Operator overloading in C++ initially comes off as a complicated approach, but once you spot the general pattern you will understand how we do this. To start with we will declare a single function that will define some behaviour for us. We will then move onto defining these as methods in our classes.

### Declaring Operator Overload Functions

To declare an operator overload function we use the `operator` keyword. We also need to specify the operator we are overloading. For example, if we had a type `blob` and wanted to override the `+` operator we would do the following:

```
blob operator+(const blob &lhs, const blob &rhs)
```

This will allow us to make a call similar to that shown below:

```
1 blob a;  
2 blob b;  
3 blob c = a + b;
```

The function will return an object of type `blob` (the result of adding two blobs together) and will be called whenever an addition is invoked on two `blob` objects.

### Types of Operator

A more concise list of operators that can be overloaded is provided at [C++ Reference](#). However, the broad categories are as follows:

**arithmetic** - plus, minus, etc.

**assignment** - what happens when you attempt to assign the value of one object to another

**increment & decrement** - both pre- and post-fix

**logical** - and, or, etc.

**comparison** - equality, greater than, less than, etc.

**member access** - such as used in `vector`

**other** - casting, `new`, `delete`, etc.

We will only look at a few in this unit, and most are normally required in special cases. However, having an idea of what is possible is useful.

As an example application, consider the following:

```
1 #include <iostream>  
2  
3 using namespace std;  
4  
5 // Simple struct to test  
6 struct my_struct  
7 {  
8     unsigned int x;  
9 };  
10  
11 // Overload addition operator on my_struct  
12 my_struct operator+(const my_struct &lhs, const my_struct &rhs)  
13 {  
14     // Create a new my_struct object
```



```

15     my_struct temp;
16     // Set the x value
17     temp.x = lhs.x + rhs.x;
18     // Return temp
19     return temp;
20 }
21
22 int main(int argc, char **argv)
23 {
24     // Create two my_struct objects
25     my_struct a;
26     my_struct b;
27
28     // Initialise the x parameter
29     a.x = 10;
30     b.x = 20;
31
32     // Print details
33     cout << "a.x = " << a.x << endl;
34     cout << "b.x = " << b.x << endl;
35
36     // Add the two values together
37     auto c = a + b;
38
39     // Print details
40     cout << "c.x = " << c.x << endl;
41
42     return 0;
43 }

```

Listing 9.1: Operator Overload Function

Our override operation is defined from line 12. It takes two parameters of type `my_struct` and also returns a value of type `my_struct`. The function itself simply creates a new `my_struct` object and sets its `x` value to the sum of the two parameters' `x` value.

On line 37 we actually invoke the use of the addition operator. Notice it behaves just as a normal addition operation. The output from this application is shown below:

```

a.x = 10
b.x = 20
c.x = 30

```

Listing 9.2: Output from Operator Overload Application

As expected, the `x` value of `c` is just `a.x + b.x`.

## 9.2 Operator Overloading in Classes

Declaring an operator overload in this manner is not the standard approach, particularly because we are working in object-oriented applications with C++. Let us now look at how we define operator overload methods in classes - as you can guess it is pretty much the same as defining a standalone function. The code below illustrates the same application as above but with the operator overload implemented as a method.

**Declaring an Operator Overload Method in a class / struct**

The main difference with defining an operator overload function and a method is the parameters we pass. Consider that when we are adding one object to another we are calling the method on the object:

`a = b + c`

We are just calling the `+` operator method on `b` above, passing the parameter `c`. Therefore, we declare our operator overload method as follows:

`type operator+(const type &rhs)`

```
1 #include <iostream>
2
3 using namespace std;
4
5 // Simple struct to test
6 struct my_struct
7 {
8     unsigned int x;
9
10     my_struct operator+(const my_struct &rhs)
11     {
12         // Create value to return
13         my_struct temp;
14         // Add x together and set in x
15         temp.x = this->x + rhs.x;
16         // Return value
17         return temp;
18     }
19 };
20
21 int main(int argc, char **argv)
22 {
23     // Create two my_struct objects
24     my_struct a;
25     my_struct b;
26
27     // Initialise the x parameter
28     a.x = 10;
29     b.x = 20;
30
31     // Print details
32     cout << "a.x = " << a.x << endl;
33     cout << "b.x = " << b.x << endl;
34
35     // Add the two values together
36     auto c = a + b;
37
38     // Print details
39     cout << "c.x = " << c.x << endl;
40
41     return 0;
42 }
```

Listing 9.3: Operator Overloading as a Method

Our operator overload method is declared on line 10. It follows the same basic premise but uses the `this` pointer to access the *left-hand side* of the expression. On line 36 we call this method. The output is the same as before:

```
a.x = 10
b.x = 20
c.x = 30
```

Listing 9.4: Output from Class Based Operator Overload

## 9.3 A vec2 Class

For the rest of this unit we will build up a class with a collection of overloaded operators. The type of this class is `vec2`. A `vec2` is a *vector* (in mathematical terms) in 2D space. You can consider it as a 2D point in space.

A `vec2` is an interesting construct to examine as it allows us to explore a number of different operators. It is also a useful type for anyone working in areas such as computer graphics, physics simulations and artificial intelligence.

### What is a Vector (not the C++ type)?

A vector in broad terms represents a direction and distance in a particular space. For example, in a 3D space a vector has 3 values -  $x$ ,  $y$  and  $z$ . In 2D space we have 2 values -  $x$  and  $y$ .

You can consider a vector as a point in 2D or 3D if that helps. It is not strictly true - a vector is a direction and magnitude. This will become clearer as we work through this unit. However, the point here is to develop a class with a collection of operator overloads, so you don't have to worry so much about what the object is doing.

You need to avoid confusing a mathematical vector with the C++ `vector` type (although they are related). A `vector` stores a collection of indexed values, a mathematical vector is used in physics based calculations to represent direction and motion.

To get started, the following code defines our initial `vec2` struct and a main function to test it.

```
1 #include <iostream>
2
3 using namespace std;
4
5 // A simple vec2 object
6 struct vec2
7 {
8     float x = 0.0f;
9     float y = 0.0f;
10
11     vec2(float x, float y)
12     {
13         this->x = x;
14         this->y = y;
15     }
16 };
17
18 int main(int argc, char *argv)
```

```
19 {  
20     // Create a vec2 object  
21     vec2 a(3.0f, 5.0f);  
22  
23     // Print the vec2 data  
24     cout << "a = { " << a.x << ", " << a.y << " }" << endl;  
25  
26     return 0;  
27 }
```

Listing 9.5: Simple `vec2` Class

All we have defined is the constructor for our `vec2` and its two parameters - `x` and `y` of type `float`. Running this application will output the following:

```
a = { 0, 0 }
```

Listing 9.6: Output from `vec2` Application

## 9.4 Comparison Operators

The first set of operators we will look at is comparison operators. A comparison operator is one we use to compare two values against each other. This includes equality (`==`), less than (`<`) and greater than (`>`). We will look at these different operators in turn.

### 9.4.1 `==`

Equality is the property where two objects are considered equal. This is an interesting question to answer in many regards - when are objects equal? We could say they objects are equal when they are the same object (e.g. they point to the same area in memory), or that they have a value that is equal (e.g. 5 is equal to 5).

From the point of view of our `vec2` `struct` we will consider two objects equal if both their `x` and `y` components are equal. This is the standard definition of equality of a 2D vector.

#### Declaring an Equality Method

An equality operator has to return a `bool` value. Otherwise it follows the general rule of an operator overload. An equality operator overload is as follows:

```
bool operator==(const type &rhs)
```

The updated version of our `vec2` class is shown below. The new equality operator starts on line 18. The `main` method has also been updated.

```
1 #include <iostream>  
2  
3 using namespace std;  
4  
5 struct vec2  
6 {  
7     float x = 0.0f;  
8     float y = 0.0f;
```

```

9
10 // Equality operator
11 vec2(float x, float y)
12 {
13     this->x = x;
14     this->y = y;
15 }
16
17 // Equality operator
18 bool operator==(const vec2 &rhs)
19 {
20     // Test if x and y are equal
21     return (this->x == rhs.x) && (this->y == rhs.y);
22 }
23 };
24
25 int main(int argc, char **argv)
26 {
27     // Create three vec2 objects
28     vec2 a(10.0f, 20.0f);
29     vec2 b(20.0f, 10.0f);
30     vec2 c(20.0f, 10.0f);
31
32     // Print the vec2 details
33     cout << "a = { " << a.x << ", " << a.y << " }" << endl;
34     cout << "b = { " << b.x << ", " << b.y << " }" << endl;
35     cout << "c = { " << c.x << ", " << c.y << " }" << endl;
36
37     // Check what is equal
38     cout << "a == b = " << (a == b) << endl;
39     cout << "a == c = " << (a == c) << endl;
40     cout << "b == c = " << (b == c) << endl;
41
42     return 0;
43 }

```

Listing 9.7: Overloading the == Operator

Notice that the equality operator for the `vec2` class just compares the two components individually and if they are both equal returns `true`. The output from this application is given below.

```

a = { 10, 20 }
b = { 20, 10 }
c = { 20, 10 }
a == b = 0
a == c = 0
b == c = 1

```

Listing 9.8: Output from == Operator Overload Application

Remember that C and C++ output `true` and `false` values as 1 or 0. This is why you are seeing the output above.

### 9.4.2 !=

Our next overload is the inequality operator (`!=`). This operator is the opposite of the equality operator and is therefore usually easy to write.

**Declaring an Inequality Method**

The inequality operator is similar to the equality operator. We declare it as follows:

```
bool operator!=(const type &rhs)
```

We will short cut the writing of the inequality operator in our `vec2` class by using the equality operator. The code we will write is as follows:

```
1 bool operator!=(const vec2 &rhs)
2 {
3     // Return not equal to
4     return !(*this == rhs);
5 }
```

Our technique is to test if the `rhs` object is equal to `this` (remember that is the object we are working in) and negating the result. As `this` is a pointer we have to dereference it (hence the use of `*this`).

The updated `vec2` test application is below. The inequality operator starts on line 24.

```
1 #include <iostream>
2
3 using namespace std;
4
5 struct vec2
6 {
7     float x = 0.0f;
8     float y = 0.0f;
9
10    // Equality operator
11    vec2(float x, float y)
12    {
13        this->x = x;
14        this->y = y;
15    }
16
17    bool operator==(const vec2 &rhs)
18    {
19        // Test if x and y are equal
20        return (this->x == rhs.x) && (this->y == rhs.y);
21    }
22
23    // Non-equality operator
24    bool operator!=(const vec2 &rhs)
25    {
26        // Return not equal to
27        return !(*this == rhs);
28    }
29 };
30
31 int main(int argc, char **argv)
32 {
33     // Create three vec2 objects
34     vec2 a(10.0f, 20.0f);
35     vec2 b(20.0f, 10.0f);
36     vec2 c(20.0f, 10.0f);
```

```

37
38 // Print the vec2 details
39 cout << "a = { " << a.x << ", " << a.y << " }" << endl;
40 cout << "b = { " << b.x << ", " << b.y << " }" << endl;
41 cout << "c = { " << c.x << ", " << c.y << " }" << endl;
42
43 // Check what is equal
44 cout << "a == b = " << (a == b) << endl;
45 cout << "a == c = " << (a == c) << endl;
46 cout << "b == c = " << (b == c) << endl;
47
48 // Check what is equal
49 cout << "a != b = " << (a != b) << endl;
50 cout << "a != c = " << (a != c) << endl;
51 cout << "b != c = " << (b != c) << endl;
52
53 return 0;
54 }

```

Listing 9.9: Overloading the != Operator

The output from this application is as follows.

```

a = { 10, 20 }
b = { 20, 10 }
c = { 20, 10 }
a == b = 0
a == c = 0
b == c = 1
a != b = 1
a != c = 1
b != c = 0

```

Listing 9.10: Output from != Operator Overload Application

### 9.4.3 <

Our next operator is less than (<). This is used to compare two objects and determine if the first one is less than the second one. Less than can be difficult to define for our `vec2` class so we will define what we mean by less than.

#### Less than for `vec2`

As a `vec2` can determine a position we can determine how far this point is from the origin (0, 0) using Pythagoras.

$$length^2 = x^2 + y^2$$

We can then compare this length value between two `vec2` objects. This is the technique that we will use.

#### Declaring a Less Than Operator

The less than operator is again similar to an equality operator. The following is the prototype of the method:

```
bool operator<(const type &rhs)
```

The following code shows the updated `vec2` class. The less than operator starts on line 30 and follows the approach for determining less than for a `vec2` as defined above. The `main` method has also been updated.

```
1 #include <iostream>
2
3 using namespace std;
4
5 struct vec2
6 {
7     float x = 0.0f;
8     float y = 0.0f;
9
10    // Equality operator
11    vec2(float x, float y)
12    {
13        this->x = x;
14        this->y = y;
15    }
16
17    bool operator==(const vec2 &rhs)
18    {
19        // Test if x and y are equal
20        return (this->x == rhs.x) && (this->y == rhs.y);
21    }
22
23    bool operator!=(const vec2 &rhs)
24    {
25        // Return not equal to
26        return !(*this == rhs);
27    }
28
29    // Less than operator
30    bool operator<(const vec2 &rhs)
31    {
32        // Use the squared length of the vector to determine less than
33        float my_length = (this->x * this->x) + (this->y * this->y);
34        float rhs_length = (rhs.x * rhs.x) + (rhs.y * rhs.y);
35
36        return (my_length < rhs_length);
37    }
38 };
39
40 int main(int argc, char **argv)
41 {
42     // Create three vectors
43     vec2 a(10.0f, 20.0f);
44     vec2 b(20.0f, 20.0f);
45     vec2 c(20.0f, 10.0f);
46
47     // Print details
48     cout << "a = { " << a.x << ", " << a.y << " }" << endl;
49     cout << "b = { " << b.x << ", " << b.y << " }" << endl;
50     cout << "c = { " << c.x << ", " << c.y << " }" << endl;
51
52     // Print less than data
53     cout << "a < b = " << (a < b) << endl;
54     cout << "a < c = " << (a < c) << endl;
55     cout << "b < c = " << (b < c) << endl;
56
57     return 0;
```



58 }

Listing 9.11: Overloading the &lt; Operator

You should compile and run this application to determine the output.

#### 9.4.4 >

Greater than is when our object is considered larger than another object. For our `vec2` this means that the length is greater than the other `vec2`'s length.

##### Declaring a Greater Than Operator

The greater than operator again follows the same basic structure of our comparison operators. This is shown below:

```
bool operator>(const type &rhs)
```

The code below provides the updated `vec2` class. The greater than operator starts on line 38. The `main` method is also updated accordingly.

```
1 #include <iostream>
2
3 using namespace std;
4
5 struct vec2
6 {
7     float x = 0.0f;
8     float y = 0.0f;
9
10    // Equality operator
11    vec2(float x, float y)
12    {
13        this->x = x;
14        this->y = y;
15    }
16
17    bool operator==(const vec2 &rhs)
18    {
19        // Test if x and y are equal
20        return (this->x == rhs.x) && (this->y == rhs.y);
21    }
22
23    bool operator!=(const vec2 &rhs)
24    {
25        // Return not equal to
26        return !(*this == rhs);
27    }
28
29    bool operator<(const vec2 &rhs)
30    {
31        // Use the squared length of the vector to determine less than
32        float my_length = (this->x * this->x) + (this->y * this->y);
33        float rhs_length = (rhs.x * rhs.x) + (rhs.y * rhs.y);
34
35        return (my_length < rhs_length);
36    }
37
```

```

38  bool operator>(const vec2 &rhs)
39  {
40      // Use the squared length of the vector to determine greater
         than
41      float my_length = (this->x * this->x) + (this->y * this->y);
42      float rhs_length = (rhs.x * rhs.x) + (rhs.y * rhs.y);
43
44      return (my_length > rhs_length);
45  }
46 };
47
48 int main(int argc, char **argv)
49 {
50     // Create three vectors
51     vec2 a(10.0f, 20.0f);
52     vec2 b(20.0f, 20.0f);
53     vec2 c(20.0f, 10.0f);
54
55     // Print details
56     cout << "a = { " << a.x << ", " << a.y << " }" << endl;
57     cout << "b = { " << b.x << ", " << b.y << " }" << endl;
58     cout << "c = { " << c.x << ", " << c.y << " }" << endl;
59
60     // Print less than data
61     cout << "a < b = " << (a < b) << endl;
62     cout << "a < c = " << (a < c) << endl;
63     cout << "b < c = " << (b < c) << endl;
64
65     // Print greater than data
66     cout << "a > b = " << (a > b) << endl;
67     cout << "a > c = " << (a > c) << endl;
68     cout << "b > c = " << (b > c) << endl;
69
70     return 0;
71 }

```

Listing 9.12: Overloading the &gt; Operator

### 9.4.5 <=

Our final two comparison operators combine the equality and less than or greater than operators. Less than or equal to (<=) compares two objects and determines if the first is less than or equal to the second.

#### Declaring a Less Than or Equal To Operator

Hopefully you are familiar with the pattern now. The less than or equal to operator is as follows:

```
bool operator<=(const type &rhs)
```

We can actually shortcut the implementation of the <= operator by reusing our greater than operator. If an object is not greater than another object, it must logically be less than or equal to. We capture this as follows:

```

1  bool operator<=(const vec2 &rhs)
2  {

```

```

3     // Just return not greater than
4     return !(*this > rhs);
5 }

```

The code below shows the updated `vec2` class and the updated main to test it.

```

1 #include <iostream>
2
3 using namespace std;
4
5 struct vec2
6 {
7     float x = 0.0f;
8     float y = 0.0f;
9
10    // Equality operator
11    vec2(float x, float y)
12    {
13        this->x = x;
14        this->y = y;
15    }
16
17    bool operator==(const vec2 &rhs)
18    {
19        // Test if x and y are equal
20        return (this->x == rhs.x) && (this->y == rhs.y);
21    }
22
23    bool operator!=(const vec2 &rhs)
24    {
25        // Return not equal to
26        return !(*this == rhs);
27    }
28
29    bool operator<(const vec2 &rhs)
30    {
31        // Use the squared length of the vector to determine less than
32        float my_length = (this->x * this->x) + (this->y * this->y);
33        float rhs_length = (rhs.x * rhs.x) + (rhs.y * rhs.y);
34
35        return (my_length < rhs_length);
36    }
37
38    bool operator>(const vec2 &rhs)
39    {
40        // Use the squared length of the vector to determine greater
41        // than
42        float my_length = (this->x * this->x) + (this->y * this->y);
43        float rhs_length = (rhs.x * rhs.x) + (rhs.y * rhs.y);
44
45        return (my_length > rhs_length);
46    }
47
48    // Less equal operator
49    bool operator<=(const vec2 &rhs)
50    {
51        // Just return not greater than
52        return !(*this > rhs);
53    }
54 };

```

```

55 int main(int argc, char **argv)
56 {
57     // Create three vectors
58     vec2 a(10.0f, 20.0f);
59     vec2 b(20.0f, 20.0f);
60     vec2 c(20.0f, 10.0f);
61
62     // Print details
63     cout << "a = { " << a.x << ", " << a.y << " }" << endl;
64     cout << "b = { " << b.x << ", " << b.y << " }" << endl;
65     cout << "c = { " << c.x << ", " << c.y << " }" << endl;
66
67     // Print less equal than data
68     cout << "a <= b = " << (a <= b) << endl;
69     cout << "a <= c = " << (a <= c) << endl;
70     cout << "b <= c = " << (b <= c) << endl;
71
72     return 0;
73 }

```

Listing 9.13: Overloading the &lt;= Operator

### 9.4.6 >=

Our final comparison operator is the greater than or equal to operator. As with the less than or equal to operator we can shortcut our implementation to being not less than.

#### Declaring a Greater Than or Equal to Operator

You should be able to guess this one by now.

```
bool operator>=(const type &rhs)
```

The updated vec2 class and main method to test it is shown below.

```

1 #include <iostream>
2
3 using namespace std;
4
5 struct vec2
6 {
7     float x = 0.0f;
8     float y = 0.0f;
9
10    // Equality operator
11    vec2(float x, float y)
12    {
13        this->x = x;
14        this->y = y;
15    }
16
17    bool operator==(const vec2 &rhs)
18    {
19        // Test if x and y are equal
20        return (this->x == rhs.x) && (this->y == rhs.y);
21    }

```

```

22
23 bool operator!=(const vec2 &rhs)
24 {
25     // Return not equal to
26     return !(*this == rhs);
27 }
28
29 bool operator<(const vec2 &rhs)
30 {
31     // Use the squared length of the vector to determine less than
32     float my_length = (this->x * this->x) + (this->y * this->y);
33     float rhs_length = (rhs.x * rhs.x) + (rhs.y * rhs.y);
34
35     return (my_length < rhs_length);
36 }
37
38 bool operator>(const vec2 &rhs)
39 {
40     // Use the squared length of the vector to determine greater
41     // than
42     float my_length = (this->x * this->x) + (this->y * this->y);
43     float rhs_length = (rhs.x * rhs.x) + (rhs.y * rhs.y);
44
45     return (my_length > rhs_length);
46 }
47
48 bool operator<=(const vec2 &rhs)
49 {
50     // Just return not greater than
51     return !(*this > rhs);
52 }
53
54 // Greater equal operator
55 bool operator>=(const vec2 &rhs)
56 {
57     // Just return not less than
58     return !(*this < rhs);
59 }
60
61 int main(int argc, char **argv)
62 {
63     // Create three vectors
64     vec2 a(10.0f, 20.0f);
65     vec2 b(20.0f, 20.0f);
66     vec2 c(20.0f, 10.0f);
67
68     // Print details
69     cout << "a = { " << a.x << ", " << a.y << " }" << endl;
70     cout << "b = { " << b.x << ", " << b.y << " }" << endl;
71     cout << "c = { " << c.x << ", " << c.y << " }" << endl;
72
73     // Print less equal than data
74     cout << "a <= b = " << (a <= b) << endl;
75     cout << "a <= c = " << (a <= c) << endl;
76     cout << "b <= c = " << (b <= c) << endl;
77
78     // Print greater equal than data
79     cout << "a >= b = " << (a >= b) << endl;
80     cout << "a >= c = " << (a >= c) << endl;

```

```
81 cout << "b >= c = " << (b >= c) << endl;  
82  
83 return 0;  
84 }
```

Listing 9.14: Overloading the &gt;= Operator

## 9.5 Arithmetic Operators

Arithmetic operators are the ones that allow us to treat our objects as values that we can add, subtract, multiple (scale), etc. This is an important concept when working in programming. Being able to work with user defined types and perform arithmetic on them is quite fundamental to larger problems. We will only look at the four basic arithmetic operators here.

### 9.5.1 Change in Approach

From now on we will only look at the method to be implemented and a `main` method to test it. This is to ensure that we don't end up with lots of code blocks. At this stage you should be able to determine how to add the code as a method to the `vec2` class.

### 9.5.2 +

Hopefully you know what addition is by now. When working with operator overloading, adding two values together makes sense for any numerical type ( $5 + 4$ ,  $2.0 + 5.555$ , etc.). However when it comes to other types we sometimes want to deal with the abstract idea of adding values together. For example, we can add two `string` objects together:

$$\text{"Hello World"} = \text{"Hello "} + \text{"World"}$$

Working with objects in this manner is an important abstraction. The approach above is no different than using the string concatenation function we used previously:

```
1 strcat("Hello ", "World");
```

However, it is much easier to understand the basic idea of joining two `string` objects together using the `+` operator:

```
1 string str = "Hello " + "World";
```

Therefore using the addition operator in this manner is more intuitive and easier to remember.

#### Declaring an Addition Operator

This is our first look at an arithmetic operator so let us spend some time thinking about this. First of all think about a simple numerical addition such as:

$$10 = 7 + 3$$

If we think about the types of these objects in computer terms we have: `int`

```
= int + int
```

So our return type from an addition operation is the same type. The type of the parameter we are adding is also the same type. For an `int` our operator overload for `+` would look as follows:

```
int operator+(const int &rhs)
```

Generalising this to a particular type we get the following: *type*

```
operator+(const type &rhs)
```

### Adding Two Vectors

Vector addition requires us to add each of the individual components of the vector together. For example, if we consider a 2-dimensional vector as follows:

$$(v_x, v_y)$$

Then adding two vectors,  $v$  and  $u$ , into the vector  $w$  requires us to perform the following calculation:

$$w_x = v_x + u_x$$

$$w_y = v_y + u_y$$

This is what we will implement in our `vec2` add operator.

The addition operator overload for our `vec2` class is given below. *This needs to be added to your existing `vec2` class.*

```
1 // Add operator
2 vec2 operator+(const vec2 &rhs)
3 {
4     // Add the components
5     return vec2(this->x + rhs.x, this->y + rhs.y);
6 }
```

Listing 9.15: + Operator Overload Function

Notice that we just return another `vec2` object with the two components of the `vec2` objects being added. We again use the `this` pointer to access the attributes of the local object.

The `main` test application for this new operator is given below. Here we are adding three different `vec2` objects together in different ways and then displaying the results. Notice that we can just use the `+` operator to add the `vec2` objects together and store the results.

```
1 int main(int argc, char **argv)
2 {
3     // Define 3 vec2 objects
4     vec2 a(10.0f, 20.0f);
5     vec2 b(20.0f, 10.0f);
6     vec2 c(20.0f, 20.0f);
7 }
```

```

8 // Print the values
9 cout << "a = { " << a.x << ", " << a.y << " }" << endl;
10 cout << "b = { " << b.x << ", " << b.y << " }" << endl;
11 cout << "c = { " << c.x << ", " << c.y << " }" << endl;
12
13 // Add some vec2 together
14 auto d = a + b;
15 auto e = a + c;
16 auto f = b + c;
17
18 // Print the values
19 cout << "a + b = { " << d.x << ", " << d.y << " }" << endl;
20 cout << "a + c = { " << e.x << ", " << e.y << " }" << endl;
21 cout << "b + c = { " << f.x << ", " << f.y << " }" << endl;
22
23 return 0;
24 }

```

Listing 9.16: Test Application for + Operator Overload

Running this application will give you an output as follows:

```

a = { 10, 20 }
b = { 20, 10 }
c = { 20, 20 }
a + b = { 30, 30 }
a + c = { 30, 40 }
b + c = { 40, 30 }

```

Listing 9.17: Output from + Operator Overload Application

*Make sure you understand the result. This is fundamental for your understanding of how the operator overload has worked.*

### 9.5.3 -

The subtraction operator (-) allows us to subtract one object from another. This can sometimes be trickier to understand for different object types. For example, does it make sense to write code such the following?

```
‘‘Hello’’ - ‘‘World’’
```

Probably not, and in C++ you cannot subtract `string` objects from each other. However, for mathematical types it does normally make sense to have some form of subtraction, and our `vec2` type is no different.

#### Declaring a Subtraction Operator

The - operator follows the previous pattern we saw for the addition operator. Remember that when we defined an addition originally in mathematical terms. We can do the same for subtraction:

$$10 = 15 - 5$$

As types we have:

```
int = int - int
```

So we can define our operator overload for subtraction as follows:



```
type operator-(const type &rhs)
```

### Subtracting Two Vectors

Subtraction of two vectors follows the same premise as we defined previously:

$$w_x = v_x - u_x$$

$$w_y = v_y - u_y$$

The operator overload for subtraction in our `vec2` class is given below. Notice the similarity between this method and the addition one.

```
1 // Subtraction operator
2 vec2 operator-(const vec2 &rhs)
3 {
4     // Sub the components
5     return vec2(this->x - rhs.x, this->y - rhs.y);
6 }
```

Listing 9.18: - Operator Overload Function

Our main test application is similar to our application for testing addition but instead subtracts vectors from each other. The `main` function is below:

```
1 int main(int argc, char **argv)
2 {
3     // Define 3 vec2 objects
4     vec2 a(10.0f, 20.0f);
5     vec2 b(20.0f, 10.0f);
6     vec2 c(20.0f, 20.0f);
7
8     // Print the values
9     cout << "a = { " << a.x << ", " << a.y << " }" << endl;
10    cout << "b = { " << b.x << ", " << b.y << " }" << endl;
11    cout << "c = { " << c.x << ", " << c.y << " }" << endl;
12
13    // Subtract some vec2 together
14    auto d = a - b;
15    auto e = a - c;
16    auto f = b - c;
17
18    // Print the values
19    cout << "a - b = { " << d.x << ", " << d.y << " }" << endl;
20    cout << "a - c = { " << e.x << ", " << e.y << " }" << endl;
21    cout << "b - c = { " << f.x << ", " << f.y << " }" << endl;
22
23    return 0;
24 }
```

Listing 9.19: Test Application for - Operator Overload

On lines 14 to 16 we perform the actual subtraction between `vec2` objects. The result from running this application is as follows.

```
a = { 10, 20 }
b = { 20, 10 }
c = { 20, 20 }
a - b = { -10, 10 }
a - c = { -10, 0 }
b - c = { 0, -10 }
```

Listing 9.20: Output from – Operator Overload Application

*Again ensure that you understand the how this code operates. Spend time stepping through it by hand to ensure you comprehend the individual pieces.*

### 9.5.4 \*

Our next operator overload is multiplication (\*). As with subtraction, understanding what we mean by multiplication of a type can be difficult. For example, in a programming language such as Python it is quite common to write something like:

```
str = 3 * 'Hello'
```

This will put ‘HelloHelloHello’ into the `str` variable. However, in C++ this is not behaviour you should expect. The multiplication of a `string` in C++ is *not defined*.

For a vector (like our `vec2` object) we have another problem - *what type of multiplication do we mean?* A vector has *three* types of multiplication possible between two vectors, and also the capability to be multiplied by a single numerical value. This latter approach is the one we will take - scaling a value.

#### Declaring Multiplication Operators

A multiplication operator can take various forms depending on the type of multiplication you support. Our approach will support scaling. This takes the following form:

```
type operator*(float scale)
```

#### Scaling a Vector

Scaling a vector involves us multiplying each component by the scalar value. The general equation is as follows:

$$v * s = (v_x * s, v_y * s)$$

So, for example, if we have a vector (10, 20) and scale it by 5 we get:

$$(10, 20) * 5 = (50, 100)$$

The operator overload for scalar multiplication for our `vec2` class is below. You should add this to your existing code now.

```
1 // Scale operator using float
2 vec2 operator*(float scale)
3 {
```

```

4     // Scale components and return
5     return vec2(this->x * scale, this->y * scale);
6 }

```

Listing 9.21: \* Operator Overload Function

Our main application for our code is below. We are still using the same base `vec2` objects but now we are scaling them.

```

1 int main(int argc, char **argv)
2 {
3     // Define 3 vec2 objects
4     vec2 a(10.0f, 20.0f);
5     vec2 b(20.0f, 10.0f);
6     vec2 c(20.0f, 20.0f);
7
8     // Print the values
9     cout << "a = { " << a.x << ", " << a.y << " }" << endl;
10    cout << "b = { " << b.x << ", " << b.y << " }" << endl;
11    cout << "c = { " << c.x << ", " << c.y << " }" << endl;
12
13    // Scale the vectors
14    auto d = a * 5;
15    auto e = b * 10;
16    auto f = c * 20;
17
18    // Print the values
19    cout << "a * 5 = { " << d.x << ", " << d.y << " }" << endl;
20    cout << "b * 10 = { " << e.x << ", " << e.y << " }" << endl;
21    cout << "c * 20 = { " << f.x << ", " << f.y << " }" << endl;
22
23    return 0;
24 }

```

Listing 9.22: Test Application for \* Operator Overload

The output from this code is shown below. *Again ensure you understand why we get the output that we do.*

```

a = { 10, 20 }
b = { 20, 10 }
c = { 20, 20 }
a * 5 = { 50, 100 }
b * 10 = { 200, 100 }
c * 20 = { 400, 400 }

```

Listing 9.23: Output from \* Operator Overload Application

### Exercise

Another form of multiplication for a vector is *component wise*. This involves multiplying two vectors together component by component. This is defined as follows:

$$v * u = (v_x * u_x, v_y * u_y)$$

Add this multiplication operator overload to the definition of the `vec2` class.

### 9.5.5 /

Division is a similar operation to multiplication, and we will treat it as such. Most of our understanding of multiplication is simply replicated for division. We will focus on scalar division of a vector for our `vec2` class.

**Declaring Division Operators**

The division operator overload is similar to the multiplication operator:

```
type operator/(float scale)
```

**Division (Scaling) of Vectors**

The rule for scaling a vector is the same as when we worked with multiplication. The general equation is:

$$v/s = (v_x/s, v_y/s)$$

So, for example, if we have a vector (10,20) and divide it by 5 we get:

$$(10,20)/5 = (2,4)$$

Our operator overload for our `vec2` class is given below.

```
1 // Divide operator
2 vec2 operator/(float scale)
3 {
4     // Divide components and return
5     return vec2(this->x / scale, this->y / scale);
6 }
```

Listing 9.24: / Operator Overload Function

The main function to test this new operator is below:

```
1 int main(int argc, char **argv)
2 {
3     // Define 3 vec2 objects
4     vec2 a(10.0f, 20.0f);
5     vec2 b(20.0f, 10.0f);
6     vec2 c(20.0f, 20.0f);
7
8     // Print the values
9     cout << "a = { " << a.x << ", " << a.y << " }" << endl;
10    cout << "b = { " << b.x << ", " << b.y << " }" << endl;
11    cout << "c = { " << c.x << ", " << c.y << " }" << endl;
12
13    // Scale the vectors
14    auto d = a / 5;
15    auto e = b / 10;
16    auto f = c / 20;
17
18    // Print the values
19    cout << "a / 5 = { " << d.x << ", " << d.y << " }" << endl;
20    cout << "b / 10 = { " << e.x << ", " << e.y << " }" << endl;
21    cout << "c / 20 = { " << f.x << ", " << f.y << " }" << endl;
22
23    return 0;
24 }
```

Listing 9.25: Test Application for / Operator Overload

Finally the output from this application is below. *Yet again, ensure you understand how the output has been generated.*

```
a = { 10, 20 }
b = { 20, 10 }
c = { 20, 20 }
a / 5 = { 2, 4 }
b / 10 = { 2, 1 }
c / 20 = { 1, 1 }
```

Listing 9.26: Output from / Operator Overload Application

### Exercise

As with the multiplication approach, implement a component wise division operator. This is defined as follows:

$$v/u = (v_x/u_x, v_y/u_y)$$

Add this operator overload to your `vec2` class definition.

## 9.6 Assignment Operators

The next two operators we will look at are known as assignment operators. Assignment operators are called when we perform such an action as assigning one value to another. For example, if we have a piece of code like this:

```
1 widget w;
2 w = 500;
```

Then an assignment operator for the `widget` class is called when we have the line `w = 500`. The assignment operator that is called is the one that matches the type (here some form of number, e.g. `int`, `float`). If no such assignment operator exists then a compiler error occurs.

We will only look at two assignment operators `=` and `+=`. The former is the most interesting, but the latter does provide us with the understanding to be able to write the other assignment operators (such as `-=`).

### 9.6.1 =

Assignment occurs whenever we use a single equals (`=`) to assign one value to another. This is so fundamental to programming that you probably don't think about it too much. However, in C++ we can actually control what happens during this process.

There are many times when you might want to write an assignment operator that enables conversion of one type to another. This is outside the scope of this module. We will just look at what happens when we assign one `vec2` to another.

#### Declaring an Assignment Operator

Again, let us break down what we are trying to do here. First of all think about a simple assignment:

```
1 int x;
2 x = 50;
```

If we think about the types of these objects we have: `int = int`

So our return type from an assignment operation is the same type. The type of the parameter we are adding is also the same type. For an `int` our operator overload for `=` would look as follows:

```
int& operator=(const int &rhs)
```

Generalising this to a particular type we get the following: `type&`  
`operator=(const type &rhs)`

We return a reference to the object as we are returning the original object (you will see this in the `vec2` example). This avoids us creating a copy of the object on return (an expensive operation). Therefore our return type is a reference (`&`). **Note that this is one of the few times you should return a reference. Returning a reference may result in returning a reference to the stack which may be no longer valid.**

For a our `vec2` class we just need to assign the `x` and `y` components accordingly. This is illustrated in the piece of code below. This needs to be added to your `vec2` class definition.

```
1  vec2& operator=(const vec2 &rhs)
2  {
3      // Set the values of this
4      this->x = rhs.x;
5      this->y = rhs.y;
6      // Return this
7      return *this;
8  }
```

Listing 9.27: `=` Operator Overload Function

Our main function to test this new operator is below:

```
1  int main(int argc, char **argv)
2  {
3      // Define 3 vec2 objects
4      vec2 a(10.0f, 20.0f);
5      vec2 b(20.0f, 10.0f);
6      vec2 c(20.0f, 20.0f);
7
8      // Print the values
9      cout << "a = { " << a.x << ", " << a.y << " }" << endl;
10     cout << "b = { " << b.x << ", " << b.y << " }" << endl;
11     cout << "c = { " << c.x << ", " << c.y << " }" << endl;
12
13     // Assign values
14     auto d = b;
15     auto e = c;
16     auto f = a;
17
18     // Print the values
19     cout << "d = { " << d.x << ", " << d.y << " }" << endl;
20     cout << "e = { " << e.x << ", " << e.y << " }" << endl;
```

```

21     cout << "f = { " << f.x << ", " << f.y << " }" << endl;
22
23     return 0;
24 }

```

Listing 9.28: Test Application for = Operator Overload

Running this application will provide the following output:

```

a = { 10, 20 }
b = { 20, 10 }
c = { 20, 20 }
d = { 20, 10 }
e = { 20, 20 }
f = { 10, 20 }

```

Listing 9.29: Output from = Operator Overload Application

Overloading the assignment operator brings back the issue of how objects are constructed and destructed again. Let us look at this once again.

### Construction and Destruction (yet again!)

Let us consider a situation as follows:

```

1 vec2 x(1, 2);
2 vec2 y(3, 4);
3 x = y;
4 y = vec2(5, 6);

```

The calls that will be made in your application will be as follows:

**line 1** - constructor for `vec2` called

**line 2** - constructor for `vec2` called

**line 3** - assignment operator for `vec2` called

**line 4** - constructor for `vec2` called, assignment operator for `vec2` called, then destructor for `vec2` called.

**end of application** - destructor for `vec2` called twice

Line 4 is the interesting one which you might find hard to agree with straight away, but think of it like this. The `vec2` object (5,6) has to be constructed (step 1). We then assign this to `y` (step 2). After the assignment, the `vec2` object we created is no longer needed. Therefore it is destroyed. Therefore, for that little line of code, three separate function calls are made - quite a bit if we do it too often.

### 9.6.2 +=

Our next operator overload is `+=`. The `+=` operator is one that you may be less familiar with. It is used as a shorthand method of adding one value to another, storing the result in the former. For example, if we have the line of code:

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

This can be reduced to the following:

```
x += y;
```

Doing this allows us to combine two operations in one. It is also less typing! Other similar operators exist:

- `--`
- `*=`
- `/=`

We are only going to focus on the `+=` operator here, although you should be able to work out how to implement the others from the example.

### Declaring a `+=` Operator

The `+=` operator looks similar to the standard assignment operator:

```
type& operator+=(const type &rhs)
```

The `+=` operator overload for our `vec2` class is below. Notice that it just combines the idea of the addition operator (`+`) and the assignment operator (`=`). Add this to your `vec2` definition now.

```
1 // Plus equals operator
2 vec2& operator+=(const vec2 &rhs)
3 {
4     // Set the values of this
5     this->x = this->x + rhs.x;
6     this->y = this->y + rhs.y;
7     // Return this
8     return *this;
9 }
```

Listing 9.30: `+=` Operator Overload Function

The main function to test this application is below.

```
1 int main(int argc, char **argv)
2 {
3     // Define 3 vec2 objects
4     vec2 a(10.0f, 20.0f);
5     vec2 b(20.0f, 10.0f);
6     vec2 c(20.0f, 20.0f);
7
8     // Print the values
9     cout << "a = { " << a.x << ", " << a.y << " }" << endl;
10    cout << "b = { " << b.x << ", " << b.y << " }" << endl;
11    cout << "c = { " << c.x << ", " << c.y << " }" << endl;
12
13    // Add b to a
14    a += b;
15
16    // Print the value
17    cout << "a + b = { " << a.x << ", " << a.y << " }" << endl;
18
19    // Add c to a
20    a += c;
```



```

21
22 // Print the value
23 cout << "a + b + c = { " << a.x << ", " << a.y << " }" << endl;
24
25 return 0;
26 }

```

Listing 9.31: Test Application for += Operator Overload

The output from this test application is below. *Again, ensure that you understand how the application comes to the results given.*

```

a = { 10, 20 }
b = { 20, 10 }
c = { 20, 20 }
a + b = { 30, 30 }
a + b + c = { 50, 50 }

```

Listing 9.32: Output from += Operator Overload Application

### Exercise

Now that you have a += operator, develop the other similar operators for `vec2`. As a reminder, these are:

- -=
- \*=
- /=

## 9.7 Complete vec2 Class

That is it for the main body of the `vec2` class. The list of operators we have overridden is provided in Table [9.1](#).

The complete code listing for the `vec2` class is below.

```

1 struct vec2
2 {
3     float x = 0.0f;
4     float y = 0.0f;
5
6     vec2(float x, float y)
7     {
8         this->x = x;
9         this->y = y;
10    }
11
12    bool operator==(const vec2 &rhs)
13    {
14        // Test if x and y are equal
15        return (this->x == rhs.x) && (this->y == rhs.y);
16    }
17
18    bool operator!=(const vec2 &rhs)
19    {
20        // Return not equal to
21        return !(*this == rhs);
22    }

```

Operator	Example	Description
==	lhs == rhs	Checks if two values are equal, returning <code>true</code> if they are and <code>false</code> otherwise.
!=	lhs != rhs	Checks if two values are equal, returning <code>false</code> if they are not and <code>true</code> otherwise.
<	lhs < rhs	Checks if lhs is less than rhs, returning <code>true</code> if it is and <code>false</code> otherwise.
>	lhs > rhs	Checks if lhs is greater than rhs, returning <code>true</code> if it is and <code>false</code> otherwise.
<=	lhs <= rhs	Checks if lhs is less than or equal to rhs, returning <code>true</code> if it is and <code>false</code> otherwise.
>=	lhs >= rhs	Checks if lhs is greater than or equal to rhs, returning <code>true</code> if it is and <code>false</code> otherwise.
+	lhs + rhs	Adds rhs to lhs.
-	lhs - rhs	Subtracts rhs from lhs.
*	lhs * s	Multiplies lhs by the scalar s.
/	lhs / s	Divides lhs by the scalar s.
=	lhs = rhs	Assigns the value of rhs to lhs.
+=	lhs += rhs	Adds rhs to lhs and assigns the added value to lhs.

Table 9.1: Operator Overloads for `vec2`

```

23
24 bool operator<(const vec2 &rhs)
25 {
26     // Use the squared length of the vector to determine less than
27     float my_length = (this->x * this->x) + (this->y * this->y);
28     float rhs_length = (rhs.x * rhs.x) + (rhs.y * rhs.y);
29
30     return (my_length < rhs_length);
31 }
32
33 bool operator>(const vec2 &rhs)
34 {
35     // Use the squared length of the vector to determine greater
36     // than
37     float my_length = (this->x * this->x) + (this->y * this->y);
38     float rhs_length = (rhs.x * rhs.x) + (rhs.y * rhs.y);
39
40     return (my_length > rhs_length);
41 }
42
43 bool operator<=(const vec2 &rhs)
44 {
45     // Just return not greater than
46     return !(*this > rhs);
47 }
48
49 bool operator>=(const vec2 &rhs)
50 {
51     // Just return not less than
52     return !(*this < rhs);
53 }
54
55 vec2 operator+(const vec2 &rhs)

```

```

55 {
56     // Add the components
57     return vec2(this->x + rhs.x, this->y + rhs.y);
58 }
59
60 vec2 operator-(const vec2 &rhs)
61 {
62     // Sub the components
63     return vec2(this->x - rhs.x, this->y - rhs.y);
64 }
65
66 vec2 operator*(float scale)
67 {
68     // Scale components and return
69     return vec2(this->x * scale, this->y * scale);
70 }
71
72 vec2 operator/(float scale)
73 {
74     // Divide components and return
75     return vec2(this->x / scale, this->y / scale);
76 }
77
78 vec2& operator=(const vec2 &rhs)
79 {
80     // Set the values of this
81     this->x = rhs.x;
82     this->y = rhs.y;
83     // Return this
84     return *this;
85 }
86
87 vec2& operator+=(const vec2 &rhs)
88 {
89     // Set the values of this
90     this->x = this->x + rhs.x;
91     this->y = this->y + rhs.y;
92     // Return this
93     return *this;
94 }
95 };

```

Listing 9.33: Complete `vec2` Class

We have looked at each of these operators in turn so you should understand them individually. The additional operators you have been asked to implement should firm up that knowledge. For those of you who still want to carry on, the following few sections cover some other topics under overloaded behaviour.

## 9.8 For the Brave - Copy Constructors

So far our work in object construction has really just focused on assigning values to the object's attributes. We delved a little into what happens when we copy an object (via the assignment operator `=`), but what about when we copy an object by calling a function. For example, what happens in this code?

```

1 void do_something(widget w)
2 {
3     // .. do some work

```

```
4 }
5
6 int main(int argc, char **argv)
7 {
8     // Create a widget
9     widget w;
10    // Call do_something - pass by value (copy)
11    do_something(w);
12
13    // .. do other work
14
15    return 0;
16 }
```

When we call `do_something` above, we copy the `widget w` to the function. So what happens when we do this copy? If we haven't told our program how to behave, then it defaults to copying all the parameters (a deep copy of the object). This is not necessarily what we want.

Working with copy constructors in C++ is fairly standard in large applications. We want to have some form of control over how a large object is copied.

### Declaring a Copy Constructor

A copy constructor looks just like a normal constructor but takes a reference to an object of the same type as a parameter. It takes the following form:

```
type(const type &rhs)
```

For our `vec2` class we have the following:

```
1 // Copy constructor
2 vec2(const vec2 &other)
3 {
4     this->x = other.x;
5     this->y = other.y;
6 }
```

Listing 9.34: Copy Constructor

Notice that this is very similar to our assignment operator overload, except we are not returning the `this` value. You should add the copy constructor to your `vec2` declaration. The main function to test this new operator overload is below:

```
1 int main(int argc, char **argv)
2 {
3     // Define 3 vec2 objects
4     vec2 a(10.0f, 20.0f);
5     vec2 b(20.0f, 10.0f);
6     vec2 c(20.0f, 20.0f);
7
8     // Print the values
9     cout << "a = { " << a.x << ", " << a.y << " }" << endl;
10    cout << "b = { " << b.x << ", " << b.y << " }" << endl;
11    cout << "c = { " << c.x << ", " << c.y << " }" << endl;
12
13    // Copy values
14    vec2 d(a);
15    vec2 e(b);
```

```

16     vec2 f(c);
17
18     // Print the values
19     cout << "d = { " << d.x << ", " << d.y << " }" << endl;
20     cout << "e = { " << e.x << ", " << e.y << " }" << endl;
21     cout << "f = { " << f.x << ", " << f.y << " }" << endl;
22
23     return 0;
24 }

```

Listing 9.35: Test Application for Copy Constructor

Notice that we are calling the constructor directly on lines 14 to 16. It is more likely we would use the assignment operator in this circumstance, so this is just for illustration. Normally the copy constructor is used when calling a function using pass-by-value.

The output from this test application is as follows.

```

a = { 10, 20 }
b = { 20, 10 }
c = { 20, 20 }
d = { 10, 20 }
e = { 20, 10 }
f = { 20, 20 }

```

Listing 9.36: Output from Copy Constructor Application

### Construction and Destruction (one last time)

This is the last time we will discuss object construction and destruction in the module (I promise). With the introduction of copy construction we can finalise our discussion with one last example. Consider the code below:

```

1 void do_something(widget w)
2 {
3     // ... do some work
4 }
5
6 int main(int argc, char **argv)
7 {
8     widget w;
9     do_something(w);
10
11     // ... do some other work
12
13     return 0;
14 }

```

Let us look at how this program executes:

**line 8** - default constructor for `widget` is called.

**line 9** - copy constructor for `widget` is called.

**line 4** - `do_something` function exited. Destructor for `widget` called on copy of `widget`. Copy of `widget` is no longer valid.

**line 14** - application exits. Destructor for `widget` called on `w`.

*Understanding object construction and destruction takes time but will make you a far better programmer. Although languages such as Java and C# use a garbage collector and pass everything by reference, this doesn't mean that you can ignore these concepts. You still need to think about the resources you are using and when they will be freed.*

## 9.9 For the Brave - Conversion Operators

So we can now control what happens when we assign a value (the = operator) and what happens when we copy a value (the copy constructor). What about when we decide to convert an object to a different type? Well, we can also control this by overloading a conversion operator.

For example, consider I have an object `widget`.

```
widget w;
```

Now what happens when we decide to do something like this:

```
int x = (int)w;
```

If you don't define this behaviour then we will get a compiler error. So how do we define this behaviour? This is what the conversion operator overload is for.

### Declaring a Conversion Operator

Deriving the operator overload for conversion won't actually help us here. The general structure of the overload operator is as follows:

```
operator new-type ()
```

Pretty simple really. For our `widget` example above we would need to define an operator: `operator int()`

We will only define one conversion operator for our `vec2` class - `float*`. This will return an array of `float` values (size 2) which we can access directly. This operator overload is given below.

```
1 // Casts the vec2 to an array of float
2 operator float*()
3 {
4     float *temp = new float[2] { this->x, this->y };
5     return temp;
6 }
```

Listing 9.37: Conversion Operator Example

*This operator overload is actually very bad practice! We are returning allocated memory and we need to remember to delete it afterwards.* However for our purposes it illustrates the general idea of what we are trying to do without getting into the ability to grab memory locations where the data is stored. The `main` function to test this application is below.

```

1 int main(int argc, char **argv)
2 {
3     // Define 3 vec2 objects
4     vec2 a(10.0f, 20.0f);
5     vec2 b(20.0f, 10.0f);
6     vec2 c(20.0f, 20.0f);
7
8     // Print the values
9     cout << "a = { " << a.x << ", " << a.y << " }" << endl;
10    cout << "b = { " << b.x << ", " << b.y << " }" << endl;
11    cout << "c = { " << c.x << ", " << c.y << " }" << endl;
12
13    // Cast values to float*
14    float *d = (float*)a;
15    float *e = (float*)b;
16    float *f = (float*)c;
17
18    // Print the values
19    cout << "d = { " << d[0] << ", " << d[1] << " }" << endl;
20    cout << "e = { " << e[0] << ", " << e[1] << " }" << endl;
21    cout << "f = { " << f[0] << ", " << f[1] << " }" << endl;
22
23    // Delete the data
24    delete d;
25    delete e;
26    delete f;
27
28    return 0;
29 }

```

Listing 9.38: Conversion Operator Test Application

On lines 14 to 16 we perform the conversion. This will call the overloaded operator accordingly.

```

a = { 10, 20 }
b = { 20, 10 }
c = { 20, 20 }
d = { 10, 20 }
e = { 20, 10 }
f = { 20, 20 }

```

Listing 9.39: Output from Conversion Operation Application

## 9.10 For the Brave - Member Access

We can also overload some of the operators that allow us to access members of an object. This can be useful in some circumstances, particularly when working with data structures.

Here we will only look at the array accessor - `[]`. This is the operator that allows us to treat an object like a data store. For example, when we worked with the `vector` data store we were able to access individual members using code such as `x = v[5]`. Overloading this operator is a little more complicated than the standard ones, and the other member access operators more so.

**Declaring an Index Accessor Method**

The operator we are overloading this time is `[]`. Now, depending on the type of value being used this can get difficult. We will just consider an indexed collection using an `int` to determine where in the collection we want to access. Therefore we know that we require a parameter of type `int`. So far, we have:

```
??? operator[](int index)
```

So what is the return type? Again this depends on the type of value being used. In our example for `vec2` we will be returning a `float`:

```
float& operator[](int index)
```

This one can be difficult to grasp. Let us look at example. Let us say we have the following code:

```
1 vec2 v(10, 20);
2 float x = v[1];
```

On line 2 we are effectively making the the following call:

```
x = operator[](1);
```

The value in the square brackets goes into the `index` parameter. After this it is just a standard call.

We also return a reference to the value stored. This is so we can update the value as well. *Again this is one of those rare occurrences where we return a reference from a function.*

Our member access operator overload for `vec2` is below. Notice that we are just treating the `vec2` as an array of two values, ensuring that we access only the *0th* or *1st* element.

```
1 // Overload the subscript operator
2 float& operator[](int index)
3 {
4     // Ensure index is 0 <= x <= 1
5     assert(index >= 0 && index <= 1);
6     if (index == 0)
7         return x;
8     else
9         return y;
10 }
```

Listing 9.40: Member Access Operator Example

Our test application for this new operator is below:

```
1 int main(int argc, char **argv)
2 {
3     // Define 3 vec2 objects
4     vec2 a(10.0f, 20.0f);
5     vec2 b(20.0f, 10.0f);
6     vec2 c(20.0f, 20.0f);
7
8     // Print the values
9     cout << "a = { " << a.x << ", " << a.y << " }" << endl;
```



```

10 cout << "b = { " << b.x << ", " << b.y << " }" << endl;
11 cout << "c = { " << c.x << ", " << c.y << " }" << endl;
12
13 // Print the values using member access
14 cout << "a = { " << a[0] << ", " << a[1] << " }" << endl;
15 cout << "b = { " << b[0] << ", " << b[1] << " }" << endl;
16 cout << "c = { " << c[0] << ", " << c[1] << " }" << endl;
17
18 return 0;
19 }

```

Listing 9.41: Test Application for Member Access Operator Overload

On lines 14 to 16 we use the member access operator to print the value rather than accessing the attributes individually. The output from this application is below:

```

a = { 10, 20 }
b = { 20, 10 }
c = { 20, 20 }
a = { 10, 20 }
b = { 20, 10 }
c = { 20, 20 }

```

Listing 9.42: Output from Member Access Application

## 9.11 For the Brave - Overriding the Input and Output Operators

The final operator overloads we will look at are the input and output operators. You might ask yourself what the input and output operators are. Well we have been using these for a while - they are `>>` and `<<`, the operators we use with our input and output streams such as `cout`.

So why do we want to override these operators? Well we want to control how our object is printed and also how it is read in from a input stream. This is *very* useful when working in large applications. Java has a similar capability using the `toString` method which all objects have.

### Declaring Input and Output Operators

OK this one takes a lot of explaining so let us just look at what the input operator looks like:

```
friend istream& operator>>(istream &in, type &value)
```

Let us look at this call in stages:

`operator>>` - OK the easy one. This is the operator that we are overloading.

`istream &in` - this is the input stream we are reading from. It comes in as a parameter to the function.

`type &value` - the value that we are reading into. Note that this is a reference so that we can modify the value.

`istream&` - the return type of the function is `istream&`. Why is this? Well, once we have finished reading the necessary values from the input stream into

our object we have to return the *modified* (since we read from it) input stream back to the main application. The value we have read into was passed by reference and is updated accordingly.

**friend** - this is necessary to allow the operator to behave correctly. Friendship in object-orientation is outside the scope of this module. Friendship relates to allowing other classes the ability to access your private values. All you need to know here is that it is required.

The output operator looks almost the same but using an `ostream` type:

```
friend ostream& operator<<(ostream &out, const type &value)
```

Note this time that we pass in the value as `const`. This is because we don't need to change it.

### Inputting and Outputting a `vec2`

We now have to determine how we want our `vec2` object to appear when we print it. Well we already have - our applications have been doing this since the start. Our approach is as follows:

```
{x, y}
```

And how about when we read them in? Well, we just need to read in two values: `x y`

This is how we will implement our input and output operators.

The input and output operators for our `vec2` class are as follows:

```
1 // Input stream operator
2 friend istream& operator>>(istream &in, vec2 &value)
3 {
4     // Read from istream into x and y
5     in >> value.x >> value.y;
6     // Return the istream
7     return in;
8 }
9
10 // Output stream operator
11 friend ostream& operator<<(ostream &out, const vec2 &value)
12 {
13     // Write {x, y} to ostream
14     out << "{" << value.x << ", " << value.y << "}";
15     // Return the ostream
16     return out;
17 }
```

Listing 9.43: Overriding Stream I/O Operators

The main function to test these are below. **This new application requires you to also add a *default* constructor to the `vec2` class.** Ensure you add the default constructor (one with no parameters) as well before testing.

```

1 int main(int argc, char **argv)
2 {
3     // Declare 3 vec2 objects
4     vec2 a;
5     vec2 b;
6     vec2 c;
7
8     // Prompt for and read in values
9     cout << "a = ";
10    cin >> a;
11    cout << "b = ";
12    cin >> b;
13    cout << "c = ";
14    cin >> c;
15
16    // Print the values
17    cout << "a = " << a << endl;
18    cout << "b = " << b << endl;
19    cout << "c = " << c << endl;
20
21    return 0;
22 }

```

Listing 9.44: Test Application for Stream Override Application

As you can see this is much nicer code for printing our values and reading them in. An example output from this application is below:

```

a = 20 40
b = 10 10
c = 20 50
a = {20, 40}
b = {10, 10}
c = {20, 50}

```

Listing 9.45: Output from Stream Test Application

## 9.12 Exercise

1. Implement a `vec3` class. A `vec3` is similar to a `vec2` but has a third component - `z`. Ensure all the operators you have defined for `vec2` are also defined for `vec3`, updating as required.
2. **For the Brave** - add a function to `vec2` and `vec3` which gets the length of the vector. This is defined as

$$\sqrt{x^2 + y^2}$$

for `vec2` and

$$\sqrt{x^2 + y^2 + z^2}$$

for `vec3`.

3. **For the Brave** - add a function to `vec2` and `vec3` which gets the *dot* product of two vectors. This is defined as

$$v \cdot u = v_x u_x + v_y u_y$$

for `vec2` and

$$v \cdot u = v_x u_x + v_y u_y + v_z u_z$$

for `vec3`.

4. **For the Brave** - add a function to `vec3` which gets the cross product of two vectors. This is defined as:

$$v \times u = (v_y u_z - v_z u_y, v_z u_x - v_x u_z, v_x u_y - v_y u_x)$$

## Unit 10

# 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 from 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.

## 10.1 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.

Your 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.

## 10.2 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.

### 10.2.1 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:

```

1 #include <stdlib.h>
2
3 int main(int argc, char **argv)
4 {
5     int x;
6     int y;
7     char str[8];
8     return 0;
9 }
```

Listing 10.1: Values on the Stack

Our stack would look like the one shown in Figure 10.1. 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.

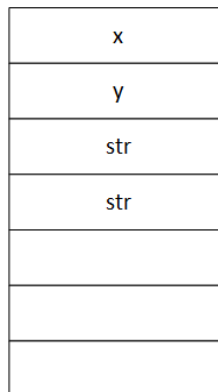


Figure 10.1: Example Stack

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?

### 10.2.2 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.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 #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 10.2: Dangerous Code Example 1 – password.c

Looking at the stack for this application we have that shown in Figure 10.2.

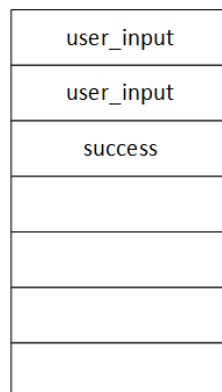


Figure 10.2: Stack for Dangerous Code Example 1

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 it 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 10.3: 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 10.4: 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 10.5: 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.

### 10.2.3 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 10.6: 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.

### 10.2.4 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 [10.3](#)

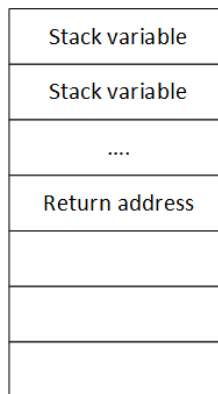


Figure 10.3: Overwriting a Return Address

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.

### 10.2.5 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.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 void copy_string(const char *input)
5 {
6     char buf[8];
7
8     // This prints out the current stack
9     printf("My stack is:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n");
10
11    // Copy raw user input into the buffer - VERY DANGEROUS!
12    strcpy(buf, input);
13    printf("%s\n", buf);
14
15    // Print the stack again
16    printf("My stack is:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n");
17 }
18

```

```

19 void uncalled()
20 {
21     printf("How was this called?!?");
22 }
23
24 int main(int argc, char **argv)
25 {
26     // We could get the function addresses using a debugger but we
27     // will cheat!
28     printf("Address of main = %p\n", main);
29     printf("Address of copy_string = %p\n", copy_string);
30     printf("Address of uncalled = %p\n", uncalled);
31
32     char input[8];
33     gets(input);
34
35     copy_string(input);
36     return 0;

```

Listing 10.7: 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 10.8: 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 10.9: 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.

## 10.3 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.

### 10.3.1 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:

```
1 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.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3
4 void modify(char *str)
5 {
6     // Try to modify the string
7     str[0] = 'm';
8 }
9
10 int main(int argc, char **argv)
11 {
12     char *str = "hello world!";

```

```
13 printf("str = %s\n", str);
14 modify(str);
15 return 0;
16 }
```

Listing 10.10: 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.

### 10.3.2 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.

### 10.3.3 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.

### 10.3.4 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.

### 10.3.5 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 providing 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:

```
1 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:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int main(int argc, char **argv)
6 {
7     wchar_t *wstr1 = L"hello world!";
8     wchar_t wstr2[13];
9     printf("%s\n", wstr1);
10    printf("%ls\n", wstr1);
11    strncpy(wstr2, wstr1, 13);
12    printf("%ls\n", wstr2);
13
14    return 0;
15 }
```

Listing 10.11: Working with Wide-character Data

If you run this you will get the following output:

```
h
hello world!
h
```

Listing 10.12: 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.

### 10.3.6 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:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int main(int argc, char **argv)
6 {
7     char buf[100];
8     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 10.13: 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 10.14: "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.

### 10.3.7 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:

```
1 if (feof(stdin)) ...
```

This will return true only if the file end point has been reached. Otherwise, an error is likely to have occurred.

### 10.3.8 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:



```

1 #include <stdio.h>
2 #include <string.h>
3
4 void func(void)
5 {
6     char buf[1024];
7
8     if (fgets(buf, sizeof(buf), stdin) == NULL)
9     {
10         /* Handle error */
11     }
12     buf[strlen(buf) - 1] = '\0';
13 }

```

Listing 10.15: 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.

### 10.3.9 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`

## 10.4 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](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.

## 10.5 Exercises

As with the previous unit, there are no exercises. You should ensure that your work is also meeting these standards.

# Unit 11

## Debugging using an Intergrated Development Environment (not assessed this year)

This non-assessed unit covers debugging applications. We will look 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 the Visual Studio Integrated Development Environment (IDE). 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 (for example, Eclipse for Java).

**On a Mac?** You can install Xcode from the App Store. This is an Integrated Development Environment for C++ on MacOS, and provides very similar debugging features. If you get stuck, the module leader can advise you further.

### 11.1 Getting Started with Visual Studio

To start off we need to get the Visual Studio IDE started. On a university machine, including the Virtual Desktop Service, use Apps Anywhere to launch it. You can sign-in using your university account when requested. On your home machine you can install Visual Studio Community 2022. You can modify your existing installation of Visual Studio Build Tools to install this by choosing Visual Studio 2022 on the Start Menu, and then Visual Studio Installer.

Once Visual Studio has started you should be presented with a screen similar to that shown in Figure [11.1](#)

Once Visual Studio has started up you need to create a new project. To do this select *Create a new project*. The window in Figure ?? will then be displayed.

From here you need to click on *Console App*. On the next screen you should enter a name for your project (e.g. Unit 08). Also note the location where the project files will be stored. Finally, note that a solution is a group of related projects.

Once the project is created your Visual Studio window should look similar to that shown in Figure [11.4](#). It may be the case that the *Solution Explorer* is on the right rather than the left shown in Figure [11.4](#)

For this first part of working with Visual Studio just use the basic "Hello World" application.

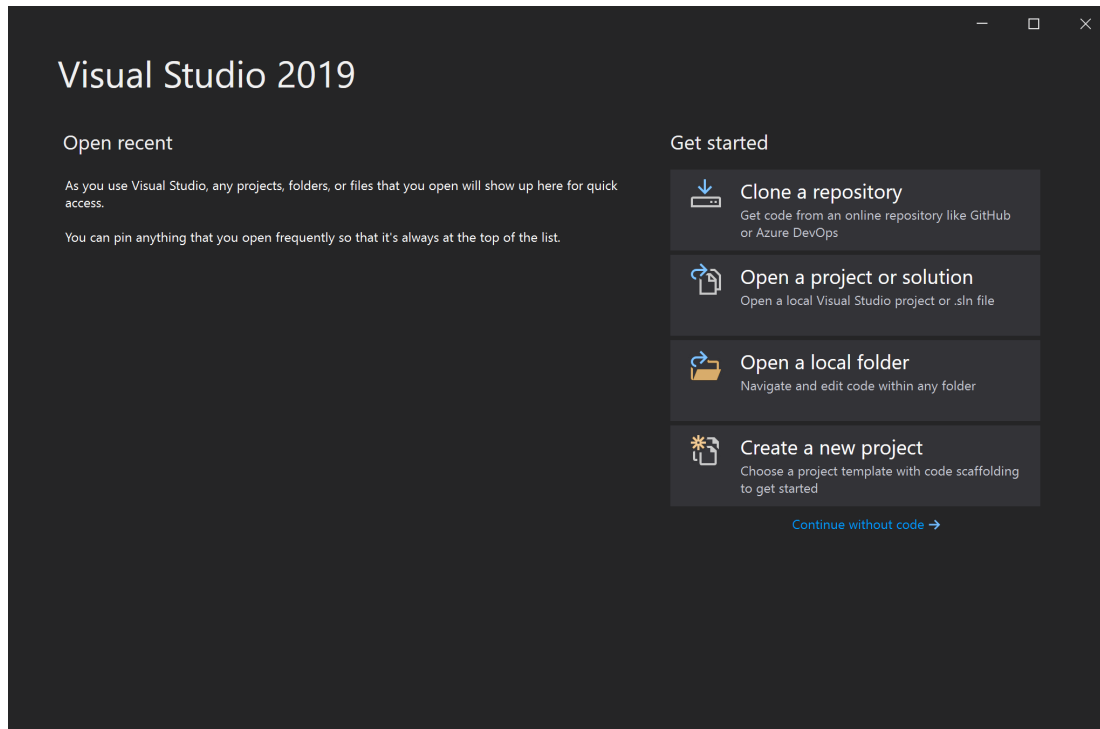


Figure 11.1: Visual Studio 2019 Start Screen

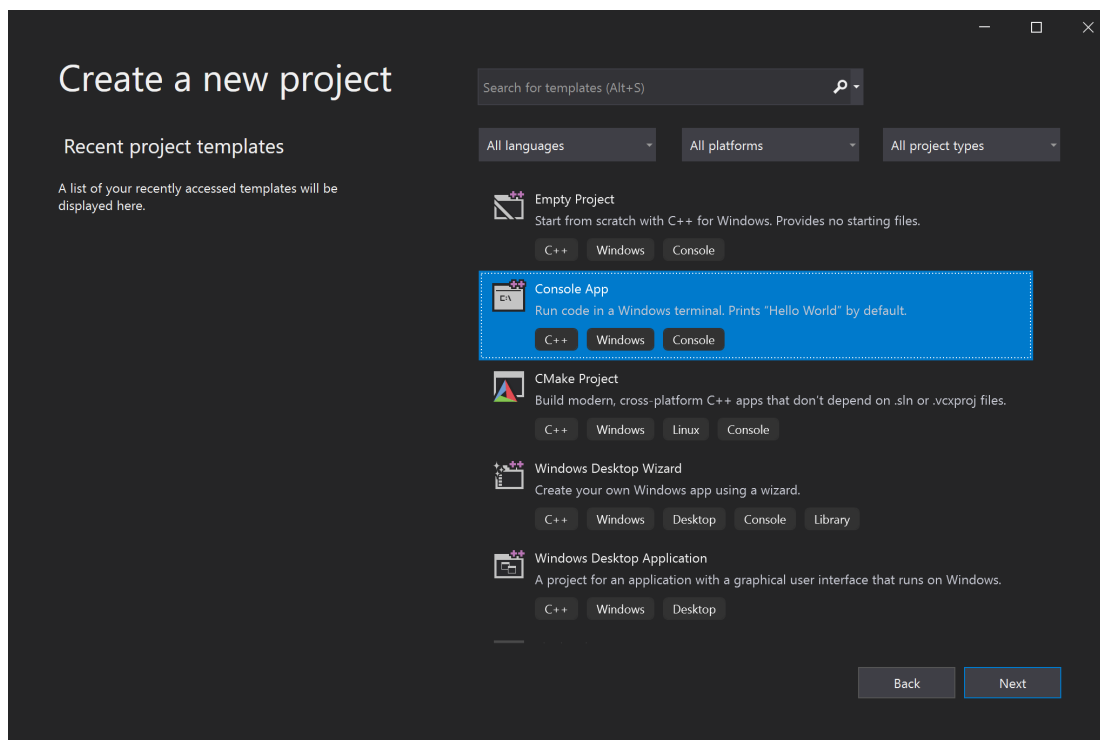


Figure 11.2: New Project Window in Visual Studio

To run your application you can press the *green start button* illustrated in Figure 11.5. 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

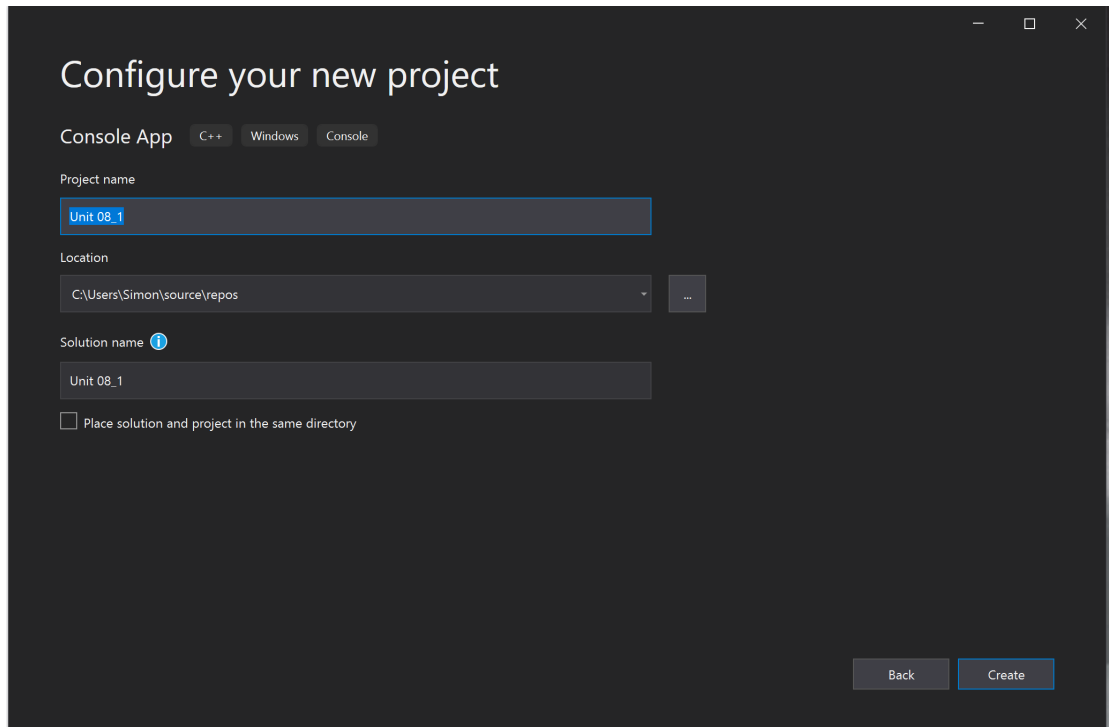


Figure 11.3: New Project Setup Configuration

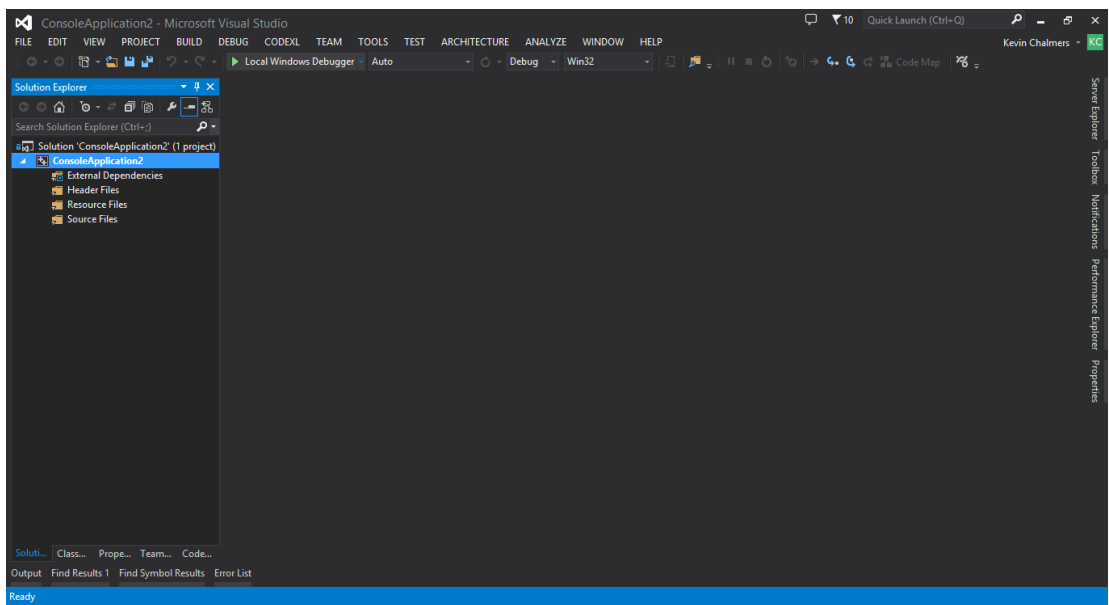


Figure 11.4: Visual Studio with Empty Project Created

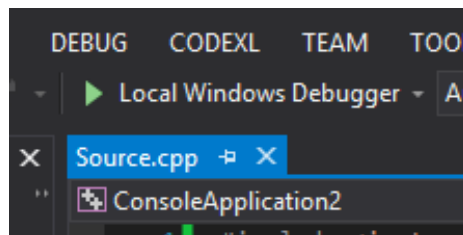


Figure 11.5: Debugging a Visual Studio Project

application exits shortly.

### 11.1.1 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 [11.6](#)

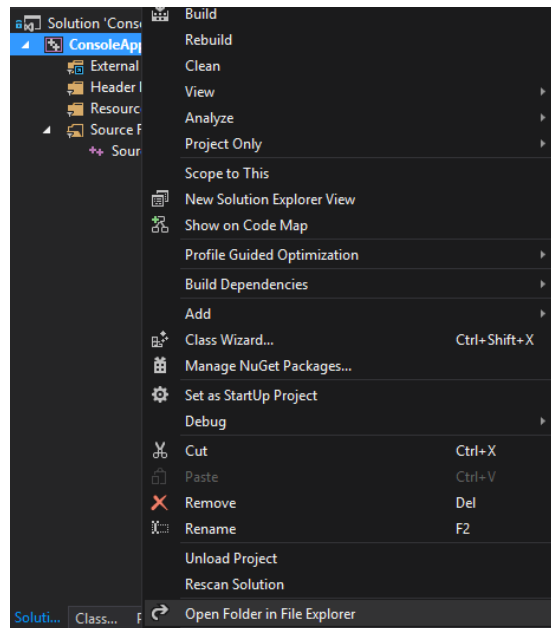


Figure 11.6: Opening Project File Location in Visual Studio

## 11.2 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 [11.7](#)

To just build (not run) the application we select **Build** ⇒ **Build Project** in the menu as shown in Figure [11.8](#)

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 [11.9](#). Do this now to see the console window staying open after it has exited.

### 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 converting a C or C++ code file to object code. Building is the act of linking the

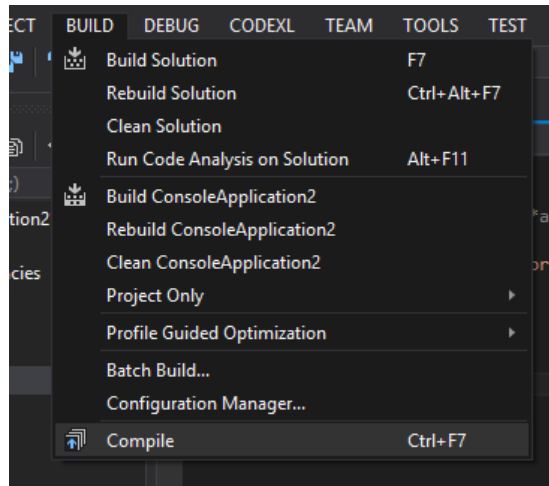


Figure 11.7: Compiling a File in Visual Studio

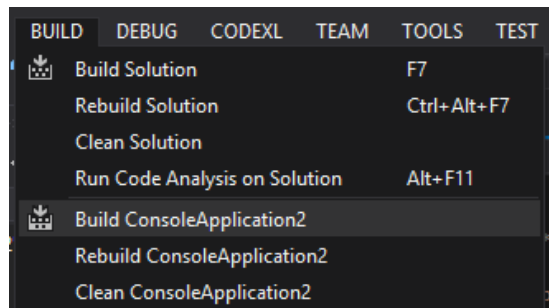


Figure 11.8: Building a Project in Visual Studio

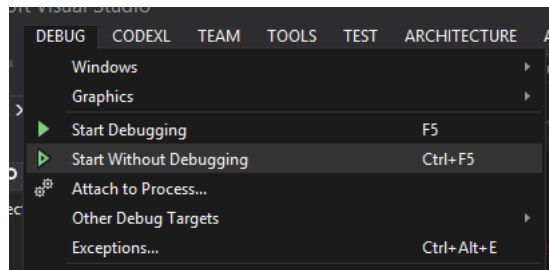


Figure 11.9: Running a Visual Studio Project without Debugging

object code files and any added libraries to create an executable. Visual Studio just has options for you to control this.

## 11.3 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 [11.10](#)

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

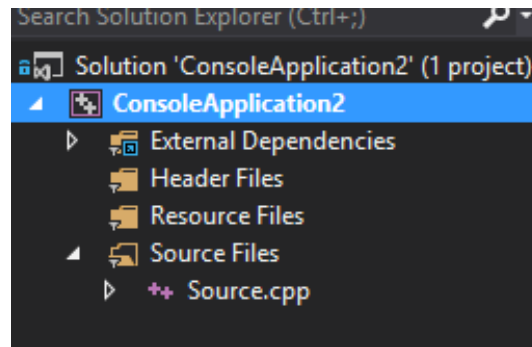


Figure 11.10: Visual Studio Project Highlighted in Solution Explorer

**Property Pages** window. This can be opened by **right clicking on the project** and selecting **Properties**. The Project Pages window is shown in Figure [11.11](#).

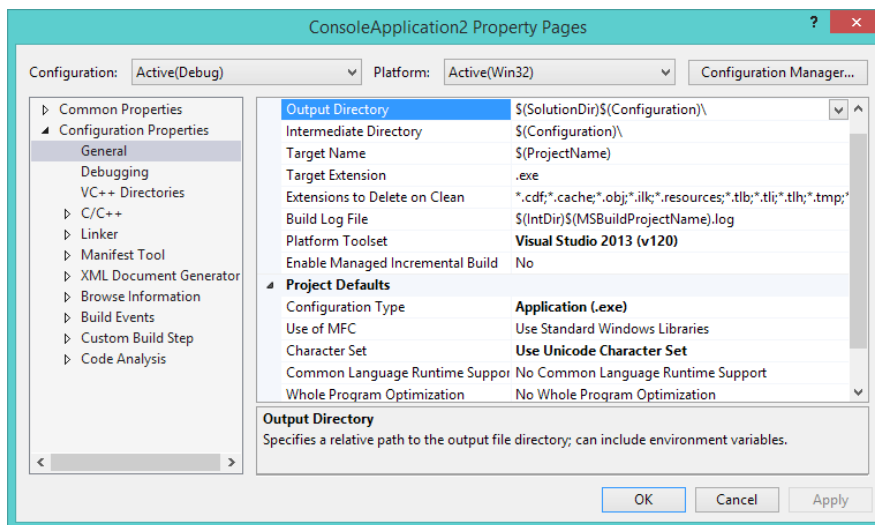


Figure 11.11: Visual Studio Project Properties

Projects themselves are contained in *Solutions*. Now let us add a new project to our solution.

## 11.4 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 [11.12](#)) the *New Project Window* will open. Ensure that you select Console App as before.

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 [11.13](#).

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.

## 11.5 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



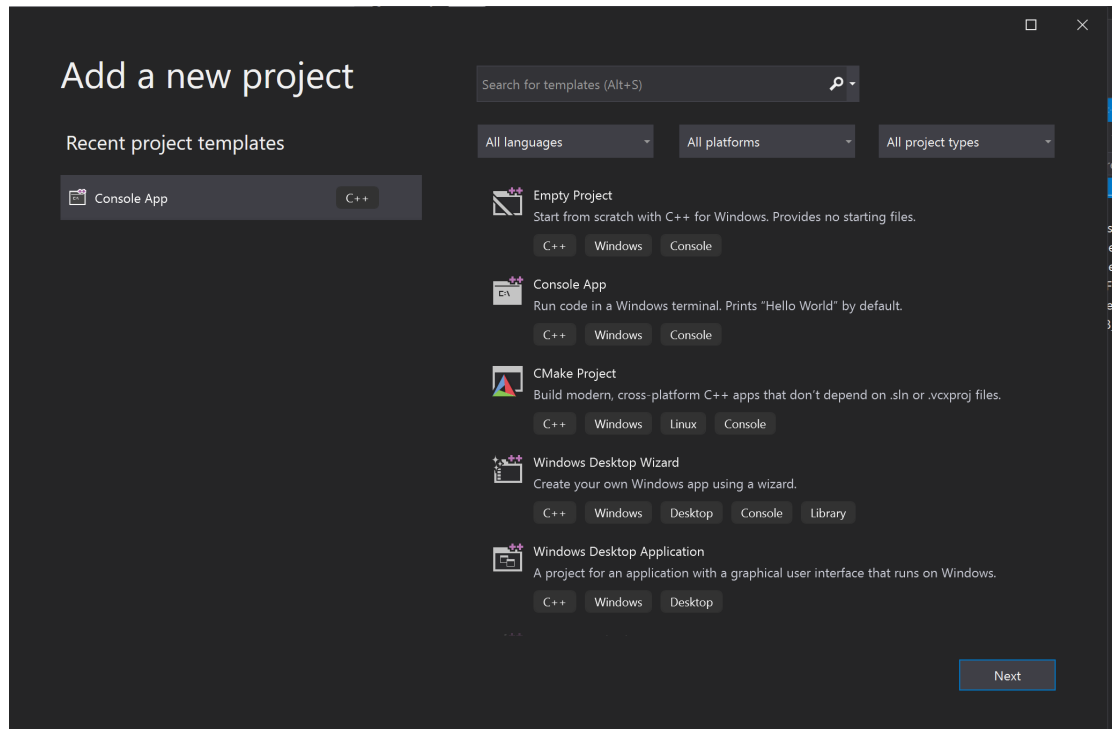


Figure 11.12: Adding a Project to an Existing Solution in Visual Studio

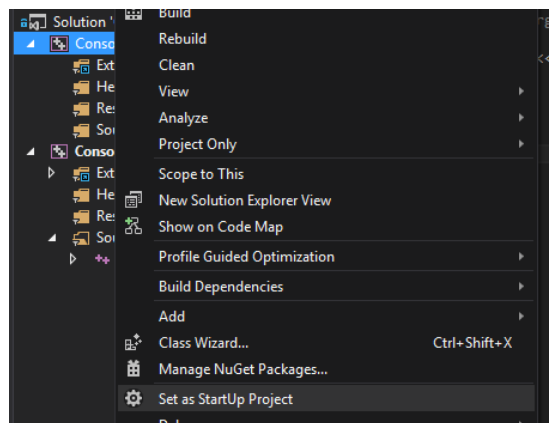


Figure 11.13: Setting the Startup Project in Visual Studio

as shown in Figure 11.14

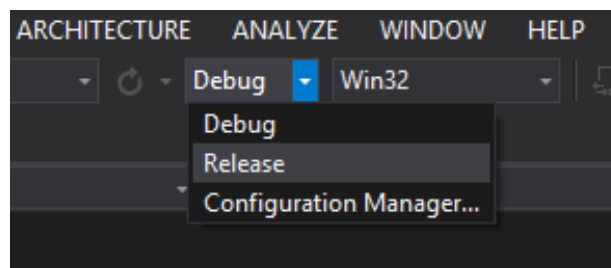


Figure 11.14: Changing Between Debug and Release Builds

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.

## 11.6 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 [11.15](#) where the red circle is the created breakpoint.

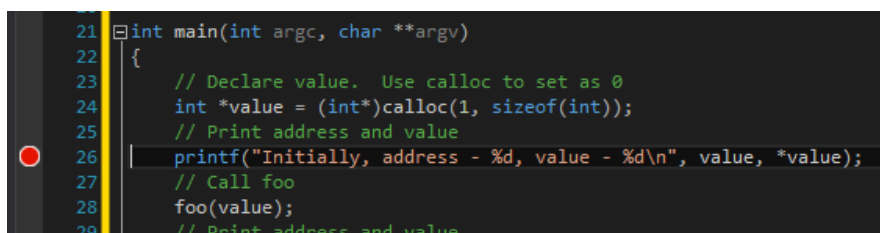


Figure 11.15: Setting a Breakpoint in Visual Studio

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 exit 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.

## 11.7 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 11.16.

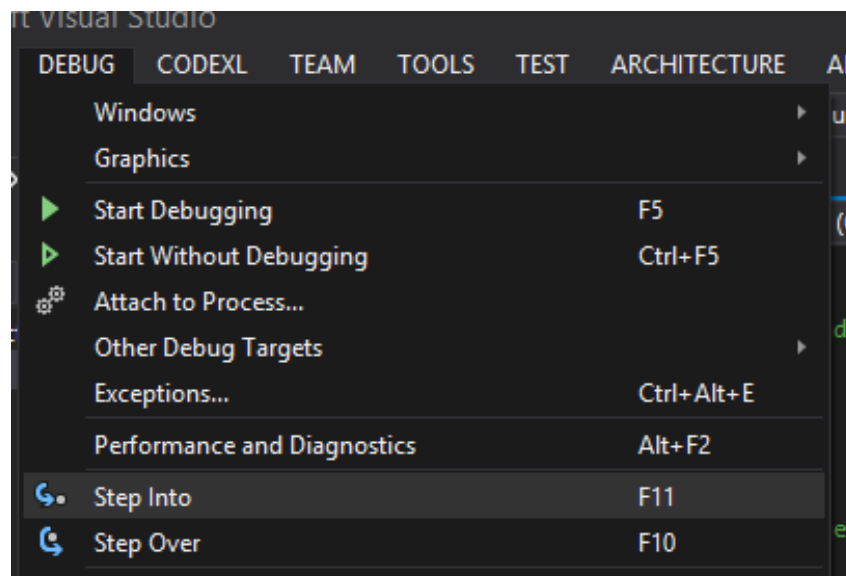


Figure 11.16: Stepping into a Program in Visual Studio

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 11.17 shows these buttons. These will become visible when the application is being debugged. On the left there is the *Find Next Statement* button. This will take you to the current line paused on. Then we have *Step Into*, *Step Over*, and *Step Out*.



Figure 11.17: Debug Stepping Tools in Visual Studio

*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.

## 11.8 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
27     // runtime
28     return 0;
29 }

```

Listing 11.1: 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 11.18.

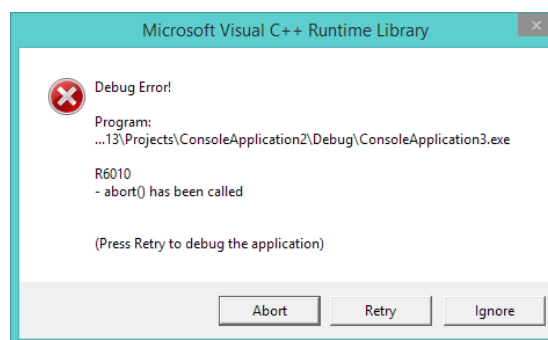


Figure 11.18: Assertion Failure in Visual Studio

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 11.2: 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.

## 11.9 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.

```
1 #include <iostream>
2
3 using namespace std;
4
5 void outermost(int &value)
6 {
7     // Multiply value by 2
8     value *= 2;
9 }
10
11 void outer(int &value)
12 {
13     // Add 5 to value
14     value += 5;
15     // Call outermost
16     outermost(value);
```

```

17 }
18
19 void inner(int &value)
20 {
21     // Divide value by 3
22     value /= 3;
23     // Call outer
24     outer(value);
25 }
26
27 void innermost(int &value)
28 {
29     // Subtract 4 from value
30     value -= 4;
31     // Call inner
32     inner(value);
33 }
34
35 int main(int argc, char **argv)
36 {
37     // Declare value - set to 100
38     int v = 100;
39     // Call innermost
40     innermost(v);
41
42     // Output value
43     cout << "Value = " << v << endl;
44
45     return 0;
46 }

```

Listing 11.3: 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 11.19 illustrates.

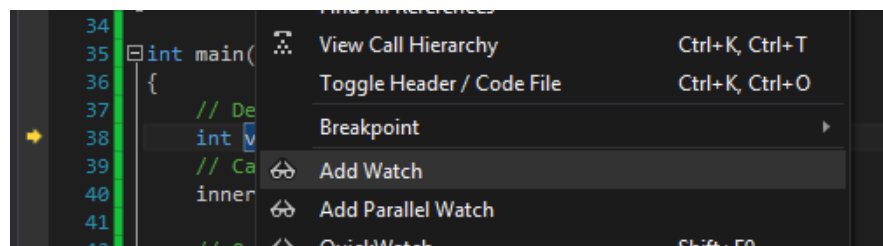


Figure 11.19: Watching a Variable in Visual Studio

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 11.20.

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.

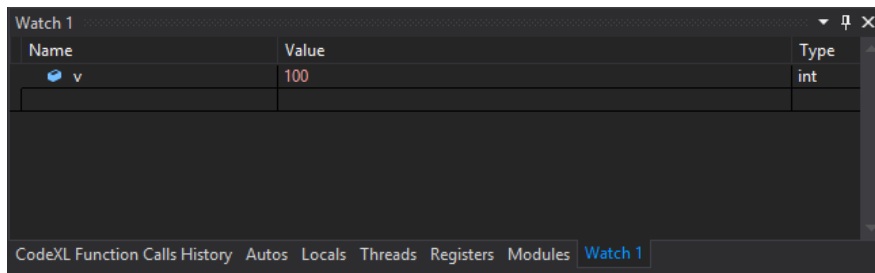


Figure 11.20: Visual Studio Watch Window

## 11.10 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.

```

1 #include <iostream>
2
3 using namespace std;
4
5 int main(int argc, char **argv)
6 {
7     // Starting value of x
8     int x = 0;
9     // Loop 100000 times
10    for (int i = 0; i < 100000; ++i)
11    {
12        // Modify x
13        x += 2;
14        x *= 2;
15    }
16
17    // Output x
18    cout << "x = " << x << endl;
19
20    return 0;
21 }

```

Listing 11.4: Advanced Breakpoints Test Application

To set a conditional breakpoint, **right click on the breakpoint** and select **Condition**. This is shown in Figure [11.21](#)

This will open the *Breakpoint Condition Window* as shown in Figure [11.22](#). We will set the condition to `x < 0`.



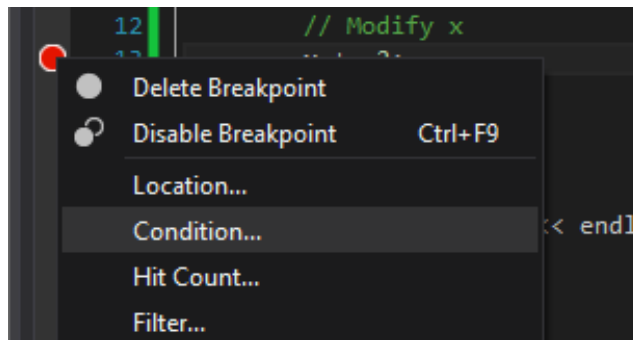


Figure 11.21: Setting an Advanced Breakpoint

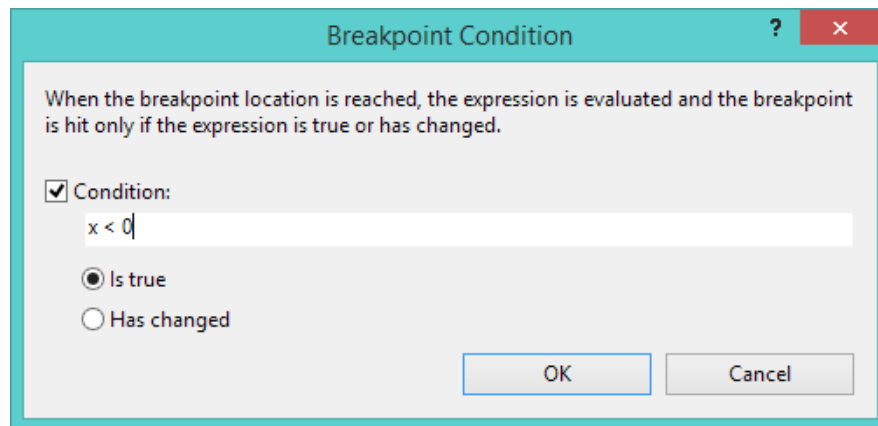


Figure 11.22: Visual Studio Breakpoint Condition Window

Try running the application and capture when `x` wraps round. What is the value of `i` at this point?

## 11.11 Examining the Call Stack

Another useful feature of debugging is the call stack. For this section you should use the *Variable Watch* code (Listing 11.3).

### 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 11.23) 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.

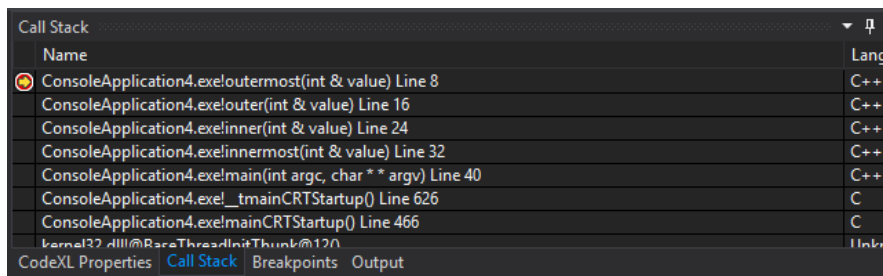


Figure 11.23: Visual Studio Call Stack

## 11.12 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 11.24 shows what this looks like in Visual Studio. We can step through these instructions using the standard commands as well.

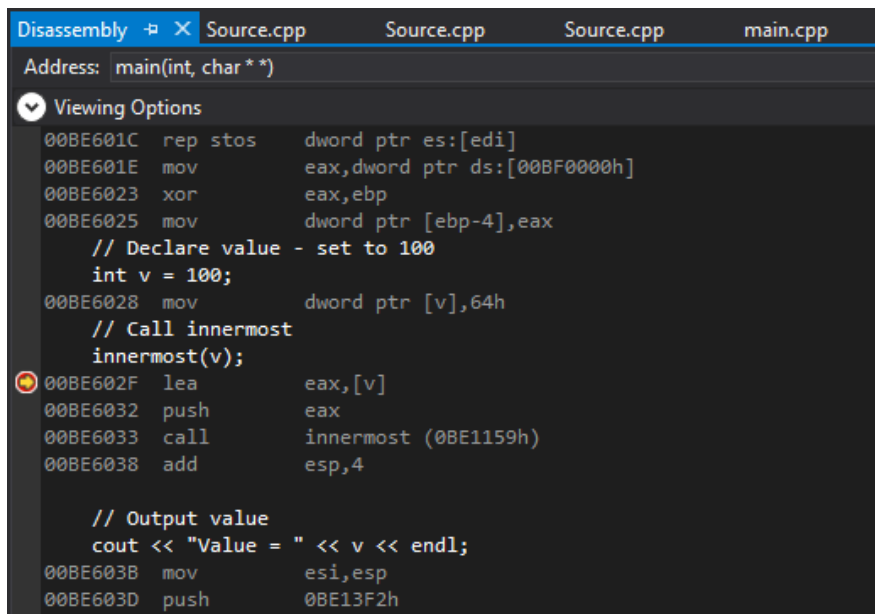


Figure 11.24: Visual Studio Disassembly

## 11.13 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 11.25 illustrates this view.

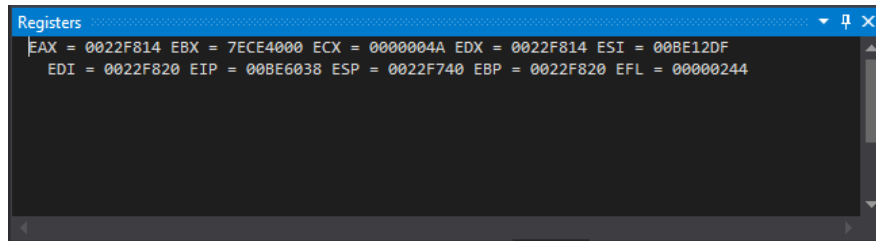


Figure 11.25: Visual Studio Registers View

## 11.14 For the Brave - Examining Memory

Visual Studio also allows you to examine raw memory (**Debug** ⇒ **Windows** ⇒ **Memory**). Figure 11.26 illustrates this view.

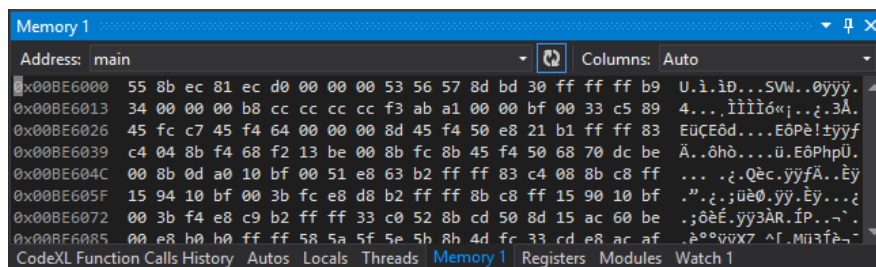


Figure 11.26: Visual Studio Memory View

## 11.15 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).

```
1 #define _CRTDBG_MAP_ALLOC
2 #include <crtdbg.h>
```

Listing 11.5: 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.

```
1 _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
```

Listing 11.6: 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.

## 11.16 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.