C++ *for Games Programming*

Coursework Module

S. Golodetz, M. Sapienza, N. Siddharth, and O. Miksik

Version of: June 9, 2016

June 6–10, Trinity Term 2016
Department of Engineering Science, University of Oxford

Contents

Chapter 1

Introduction

An ability to program in C++ is extremely useful in engineering fields such as computer vision and robotics, but there is often little time to focus in depth on practical programming skills in the course of a normal engineering degree. This course aims to fill in the gaps by teaching you:

- The basics of program development in C++.
- How to structure programs in a modular, easy-to-understand and maintainable way so that other people can build on them.
- How to use version control (specifically Git) to effectively collaborate with others.
- How to debug your programs when they go wrong.
- How to make your code and your build portable between different machines and different platforms.

Over the course of the week, the aim is to take you from a position of having limited prior programming experience to one in which you can write very simple computer games from the ground up. Games are a good target for a course such as this, because writing a game necessarily involves exercising a broad range of programming skills.

The course itself will consist of a mixture of interactive exercises to talk you through the material, and more extensive programming sessions in which you will attempt some larger mini-projects. All of the exercises that you need to attempt are highlighted and marked as follows:

Exercise 1.1: Implement a renderer for Douglas, the most accomplished of all our infinite code monkeys!

You will also find the occasional question to answer. These will be highlighted and marked as follows:

Question: What is the meaning of life? Please answer in one sentence or less.

Demonstrators will be on hand throughout the course – don't hesitate to ask them for help. These notes should be used in conjunction with the course textbook: C++ *Primer*, by S. B. Lippman et al. Unfortunately, the book is rather too comprehensive to be a real primer, so use it as a reference. Several copies are available in the lab, but you might find it advantageous to obtain your own, especially if you think you might use C++ later, e.g. for your degree project. A simpler introduction to C++ can be found in *Accelerated* C++, by Koenig and Moo.

PDF notes These notes are available as an online PDF: the demonstrators will tell you where to find this. The PDF will be useful when you want to search for particular keywords.

Course history This coursework module was developed by Stuart Golodetz, Michael Sapienza, N. Siddharth and Ondrej Miksik in 2016.

Chapter 2

Getting Started

2.1 A simple C++ program

Here is a simple C++ program that we will use to get started with compiling and running programs. The following program prints a message to the standard output.

```
1  #include <iostream> // Header file iostream contains 'std::cout'.
2
3  int main()          // All C++ programs start from a 'main' function.
4  {
5    // Write a string literal to the standard output stream.
6    std::cout << "I'm really excited to get started!\n";
7    return 0; // The return code of 0 indicates that the program succeeded.
8  }
```

In this, `std::cout` is a C++ stream object representing the standard output stream. Things that are written to it will be printed in the terminal (unless the output of the program has been redirected, e.g. to a file). Here, line 6 uses a stream insertion operator to write a string literal to `std::cout`. Both this stream insertion operator and `std::cout` are made available to the program as a result of including the `iostream` header on line 2.

2.2 Compiling and Running Programs with Visual C++

Visual C++ is an integrated development environment (IDE), rather than simply a compiler, so the first step when writing a program in it is to run the application itself. When you do this, you will initially see the window shown in Figure 2.1.

The IDE groups source files into *projects*, and projects into *solutions*[1], so the first step when starting a new program in it is to create a new solution. To do this, navigate to File → New → Project..., and select Templates → Visual C++ → Win32 → Win32 Console Application. Make sure that the 'Create directory for solution' checkbox is unticked, change the name of the project to 'gpcwmtest', and then click OK. In the dialog box that follows, click Next, then turn 'Precompiled header' and 'Security Development Lifecycle (SDL) checks' off, and 'Empty project' on, before finally clicking Finish to create the solution.

You should now see the window shown in Figure 2.2. Right click on the Source Files folder, select Add → New Item..., choose Visual C++ → Code and select C++ File (.cpp). Change the name of the file to `main.cpp` and click Add. You are now ready to write a program.

Exercise 2.1: Copy the short program shown in §2.1 into `main.cpp`. To compile it, select Build → Build Solution (or simply press the shortcut key for it, which is generally either Ctrl + Shift + B or F7). To run the program, select Debug → Start Without Debugging (usual shortcut: Ctrl + F5). If you've done it right, it should print out the message 'I'm really excited to get started!' and then exit.

[1] Presumably because it sounds suitably 'enterprise-y'.

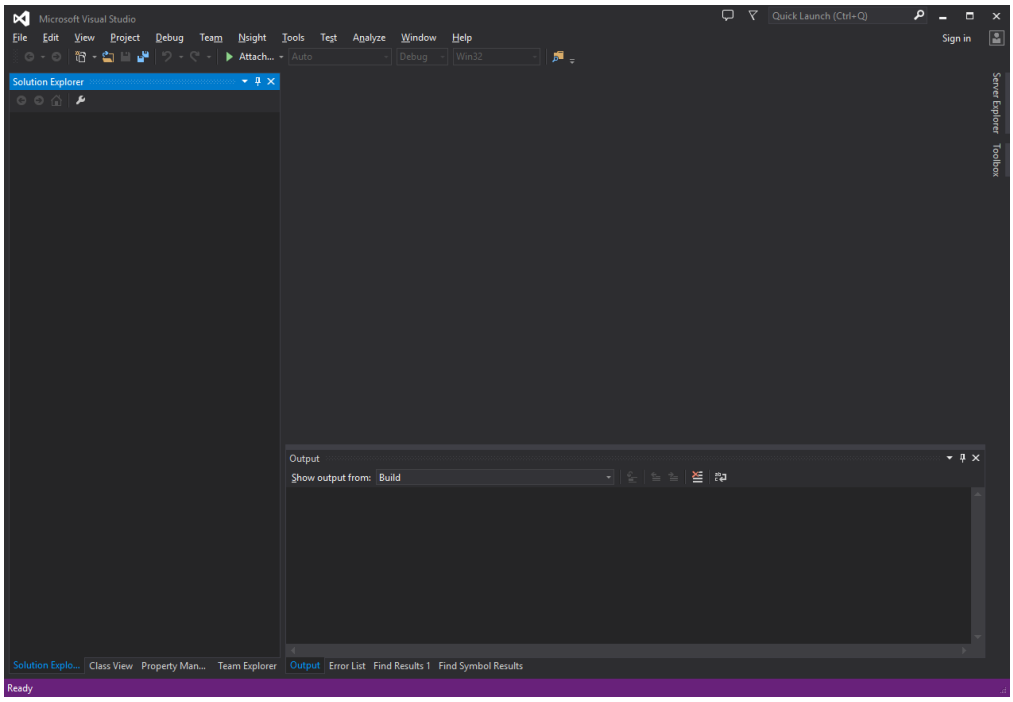


Figure 2.1: The initial situation when you open Visual C++.

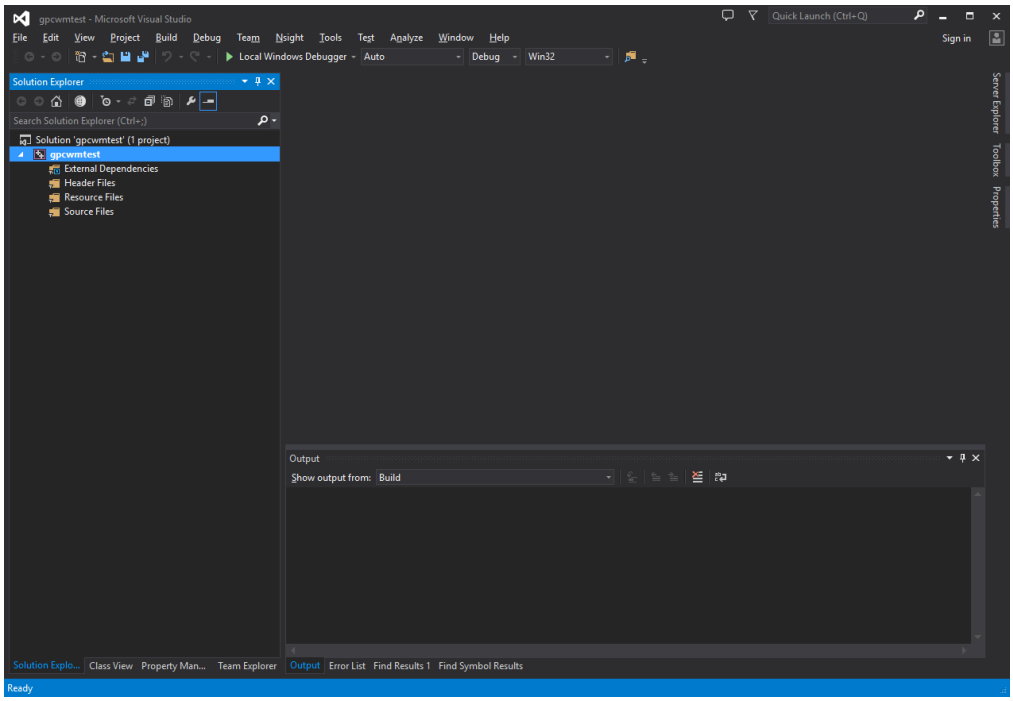


Figure 2.2: The situation after creating a new solution.

2.3 The Visual C++ Debugger

If you selected Start Without Debugging from the Debug menu in the previous section, rather than pressing the shortcut key, you may have observed that there was a menu option called Start Debugging just above it. As its name suggests, this triggers Visual C++'s debugger, which allows you to step through your program to help track down bugs: specifically, when you select this option it will run your program until either something goes wrong, or until it hits a point you have specified in the code (known as a *breakpoint*). To avoid having to set breakpoints every time you want to debug, an alternative, and useful, way to trigger the debugger is to right-click the line in your code on which you want to break and then select Run To Cursor (usual shortcut: Ctrl + F10), which runs the program until execution gets to the line in question (it acts as if you'd set a temporary breakpoint on the line in question).

Exercise 2.2: As a first experiment with the debugger, replace the program in `main.cpp` with the following and debug to the indicated line:

```
1  int main()
2  {
3    int x = 23; // debug to here
4    int y = 9;
5    if(x >= 20)
6    {
7      y = 84;
8    }
9    return 0;
10 }
```

Now look at the debugging window called Locals, which shows you the values of all the local variables in the current function (in this case, `x` and `y`). At this point, neither of these variables has been initialised, so they will both probably have weird values. However, if you 'step over' the current line by selecting Debug → Step Over (usual shortcut: F10), you will note that the value of `x` changes to 23, indicating that it has been initialised. If you continue stepping through the program, you will see the value of `y` changing after lines 4 and 7. When you get to line 9, it's sensible to stop debugging instead of continuing to step into the cleanup code that runs after `main`: to do this, select Debug → Stop Debugging (usual shortcut: Shift + F5).

Although we have not yet seen enough C++ to fully explore what the Visual C++ debugger can do for us, it is extremely powerful. As we progress through the course, you will have a chance to explore more of its capabilities, and the demonstrators will be on hand to give you some useful tips. For now, just be aware that the debugger exists, and that it's useful – we'll see a lot more of it later.

2.4 Compiling and Running Programs with g++

In §2.2, we saw how to compile and run a simple program using Microsoft's Visual C++ compiler. However, Visual C++ is generally restricted to running under Windows, so when developing on platforms such as Linux or Mac OS X, an alternative compiler must be used. One of the most popular compilers on such platforms is the GNU C++ compiler, also known as g++. In this section, you will use g++ to compile and run the simple program given in §2.1. Later in the course (see §4.3), we will see how to set up programs so that they can be built in the same way on more than one platform using the same build scripts.

Exercise 2.3: Open up VirtualBox and start the Ubuntu virtual machine. Once Ubuntu has finished booting, open up a terminal (`Ctrl + Alt + T`) and use `vi` (or your favourite editor) to create a new source file called `main.cpp`:

```
$ vi main.cpp
```

Copy the short program shown in §2.1 into this file. To compile this program into an executable that the machine can run, return to the terminal and issue the command:

```
$ g++ main.cpp -o gpcwmtest
```

Then run the executable and make sure that you can see the following message printed in the terminal:

```
$ ./gpcwmtest
I'm really excited to get started!
```

2.5 Conclusion

In this chapter, we have seen how to compile and run simple, existing programs with two different compilers: Visual C++ and g++. In the next chapter, we will look at how to write C++ programs from scratch.

2.6 Further Reading

Copious documentation for both Visual C++ and g++ can be found on the web, and we point interested students in its general direction.

Chapter 3

Basic C++ Programming

In this chapter, we present a whirlwind tour of the basic C++ programming syntax and provide exercises to get you coding quickly. This is not intended to provide a detailed explanation of all aspects of the language; for that, we refer you to the course textbook, as mentioned in Chapter 1.

3.1 Data Types and Variables

Suppose that you would like to store the numbers of fun and boring courses you are taking as variables in your program. In order to use a variable in C++, you must first declare it. Variables are declared in *variable declarations*, which consist of a type followed by a comma-separated list of variable names. For example, the following code contains a declaration that declares two variables, `funCourseCount` and `boringCourseCount`, of type `int`:

```
1 int funCourseCount, boringCourseCount;
```

Table 3.1 shows some common C++ types that you may encounter and what they represent. In addition to the built-in types provided by the language and types defined by various libraries, users can also create their own types and use them in exactly the same way as those that already exist. We will investigate how to create such types when we look at Object-Oriented Programming in Chapter 6.

In many cases (see C++ Primer for more details), variable declarations also serve as *definitions* of the variables involved. In such cases, it is possible to specify an initialising expression for each declared variable. For example:

```
1 int funCourseCount = 5, boringCourseCount = 3;
2 int totalCourseCount = funCourseCount + boringCourseCount;
```

Initialisation of variables is important, particularly for variables of primitive types such as `int` or `double`, which would otherwise be given arbitrary initial values. Note that the initialising expression for a variable does not need to be a compile-time constant, e.g. in this case `totalCourseCount` has been initialised with the result of adding the other two variables together at runtime.

Exercise 3.1: Define a few variables of different types, and print out the results. Try outputting the value of an uninitialised variable to `std::cout` and see what you get!

3.2 Conditional Branching

The simple example program we saw in §2.1 contained only a linear sequence of statements that were executed in order from start to finish. In order to create more interesting programs in which the sequence of execution can go down one of several possible paths, an `if` statement with the following syntax can be used:

Type	Represents	Necessary Includes
`bool`	Boolean (`true`/`false`)	–
`char`	Characters	–
`int`	Integers (commonly 32-bit)	–
`float`	Real numbers (single precision)	–
`double`	Real numbers (double precision)	–
`std::string`	Strings of characters	`<string>`

Table 3.1: Common C++ types that you may encounter.

```
if(conditionExpr) thenStmt [else elseStmt]
```

This evaluates the specified condition expression, and then executes `thenStmt` if and only if the condition evaluated to `true`; if the condition evaluated to `false` and an (optional) else clause was specified, `elseStmt` is executed instead of `thenStmt`. For example, the following code prints `Smile!` if `happiness` is less than 5, and `Go forth in peace!` otherwise:

```
if(happiness < 5)
  std::cout << "Smile!\n";
else
{
  std::cout << "Go forth in peace!";
  std::cout << "\n";
}
```

Note that the else clause in this case is a compound statement (a statement block) rather than an individual statement.

Another way of achieving conditional branching is using a `switch` statement, which executes one of several different blocks of code based on the value of an integral expression. For example, the following prints 'Foo' if `i == 1`, 'Bar' if `i == 2` and 'Baz' otherwise:

```
switch(i)
{
  case 1:
  {
    std::cout << "Foo\n";
    break; // really important!!!
  }
  case 2:
  {
    std::cout << "Bar\n";
    break;
  }
  default:
  {
    std::cout << "Baz\n";
    break;
  }
}
```

Observe that each `case` is terminated by a `break` in the above. This is really important, because the default behaviour in C++ is for control to 'fall through' from one `case` to the next unless the programmer specifies otherwise. Most of the time, this is not what you want, so you need to write an explicit `break` statement to prevent it.

3.3 Loops

In addition to conditional branching, we can make our programs more interesting by using loops, which allow us to specify that a particular bit of code should be run multiple times. The C++ language supports three different types of loop: `while` loops, `do` loops and `for` loops. A `while` loop has the syntax:

```
while(conditionExpr) bodyStmt
```

It repeatedly executes its body statement as long as its condition expression continues to evaluate to `true` (the condition is tested at the start of each iteration of the loop). For example, the following program prints out a countdown from 10 to 0:

```
#include <iostream>

int main()
{
  int t = 10;
  while(t >= 0)
  {
    std::cout << t << '\n';
    --t;
  }
  return 0;
}
```

A `do` loop works in much the same way as a while loop, except that the body of the loop is always executed at least once (the condition expression is evaluated at the end of each loop iteration rather than at the beginning of it). It has the following syntax:

```
do bodyStmt while(conditionExpr);
```

A `for` loop is the most general form of loop that C++ supports. It has the syntax:

```
for(initExpr; conditionExpr; loopExpr) bodyStmt
```

At the start of the loop, the initialisation expression `initExpr` is executed. Control then passes to the conditional expression `conditionExpr`, which is evaluated prior to each iteration of the loop. If the conditional expression evaluates to `true`, the body statement `bodyStmt` of the loop is then executed, followed by the loop expression `loopExpr`, after which control passes back to `condExpr`. If the conditional expression ever evaluates to `false`, the loop terminates. For example, the following program prints out a countdown from 10 to 0:

```
#include <iostream>

int main()
{
  for(int t = 10; t >= 0; --t)
  {
    std::cout << t << '\n';
  }
  return 0;
}
```

Exercise 3.2: The factorial of a (positive) integer, denoted $n!$, is the product of all positive integers less than or equal to n. (By convention, $0! = 1$.) For example, $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$. Write a program that takes as input a positive integer and outputs the factorial of that number, using variables, conditional branching and loops.

3.4 Variable Scope

Thus far, the only comment we have made about whether or not we can use a certain variable at a particular point in our program was that variables must be declared before they are used (see §3.1). However, the actual situation is somewhat more complicated than that: to properly understand what's going on, we must introduce the concept of *scope*. The scope of a variable refers to the area(s) of the program in which it can be used. Thus far, most of the variables we have seen have been declared within a statement block (enclosed by {}). These variables have *local scope*: they are accessible within the block in which they have been declared (and any blocks it encloses) at any point after the point of declaration. Their scope ends at the end of the block (i.e. at the closing curly brace), after which point they can no longer be accessed. The following example, in which each line has been annotated with the set of variables that are in scope at that point, illustrates how this works:

```
#include <iostream>

int main()
{
  int i = 23;               // {}
  int j = i + 9;            // {i}
  {
    int k = 84;             // {i,j}
    std::cout << i << '\n'; // {i,j,k}
    std::cout << k << '\n'; // {i,j,k}
  }
  std::cout << k << '\n';   // {i,j} -> error (k is not in scope)
  return 0;
}
```

Whilst it is an error to declare two variables with the same name in a single block, it is possible to declare a variable in an inner block with the same name as one in an outer block: this has the effect of *hiding* the outer variable until the inner one goes out of scope. For example:

```
int i = 23;
std::cout << i << '\n';   // prints 23
{
  int i = 9;              // hides the outer i
  std::cout << i << '\n'; // prints 9
}
std::cout << i << '\n';   // prints 23
```

In addition to declaring variables within a block, it is also possible to declare them outside of any function. Such variables have *file scope*, and are accessible anywhere within that file (technically, within that *translation unit*: see §7.2) after the point of declaration. For example:

```
int foo = 23;        // foo has file scope

int main()
{
  int bar = foo + 9; // in particular, foo can be accessed here
```

```
6   return 0;
7 }
```

It is worth noting that variables with file scope can be hidden by variables of the same name in *local scope*, just as if they had been declared in an outer block. Unlike variables declared in outer blocks though, variables at file scope can always be accessed using a technique known as *explicit qualification*: this involves prepending the variable name with an operator known as the *scope resolution operator* (written as `::`). For example, a variable `x` at file scope can always be accessed by writing `::x`.

Exercise 3.3: What will the following program output? Can you explain why?

```
1  #include <iostream>
2
3  int x = 84;
4
5  int main()
6  {
7    int y = 9;
8    {
9      int x = 11;
10     ::x = (::x / 4) + x - y;
11     std::cout << x << '\n';
12   }
13   std::cout << x << '\n';
14   return 0;
15 }
```

Variables declared in other places (for example as the parameters of functions) also have scoping rules associated with them; we will mention these as we come to them. One point that is worth making at this stage is that variables declared in `for` loop initialisers are in scope not only within the body of the loop, but also within the condition expression and the loop expression.

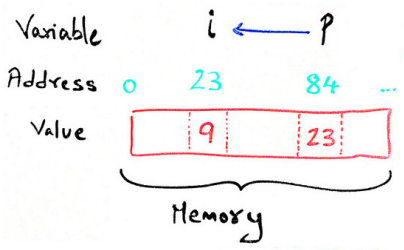
3.5 Indirection

In every program we have written up to this point, we have been referring to all of the variables directly using their names. However, when writing more complicated programs, there is sometimes a need to refer to variables indirectly (i.e. by a name other than their own). C++ gives us two related mechanisms for doing this: *pointers* and *references*.

Pointers

To understand pointers, the first important thing to understand is that the value of each variable in a program is stored at some location in memory (called an *address*). For example, the value of an `int` called `i` might really be stored at address 23. If `i` has value 9, then the memory at address 23 will have value 9, and whenever the compiler sees a mention of `i`, it will effectively replace it with something that says 'the variable at address 23'.

A *pointer* is just a variable whose value is the address of another variable. For example, we could have a pointer `p` whose own address is 84 and whose value is 23, indicating that it points to `i` (compare this to `i` itself, which has an address of 23 and a value of 9). This situation is illustrated in the following figure:



Pointer variables in C++ are just those that have a pointer type: syntactically, such types are written using an asterisk (`*`). For example, the type 'pointer to int' would be written as `int *`. To take the address of a variable, we use the *address of* operator, written as `&`. For example, the address of `i` would be written as `&i`. The situation we just described, in which `i` is an integer with value 9 and `p` is a pointer that points to `i`, can then be written in code as follows:

```
int i = 9;
int *p = &i; // p is a pointer to int whose value is the address of i
```

Given a pointer that points to a variable, it is also possible to perform the opposite of the *address of* operation just described in order to obtain a reference to the original variable. This is known as *dereferencing* the pointer, and should only be performed on pointers that actually point to variables (it is possible for pointers to point either nowhere or to an invalid address in memory). The dereferencing operator is written using an asterisk (`*`): don't confuse this with the asterisk used in the type of a pointer. For example, dereferencing `p` would be written as `*p`: this would give us back a reference to `i`. It is possible to set the value of `i` via `p` in this way. For example:

```
*p = 84; // sets i to 84
```

It is entirely possible to have multiple levels of indirection, i.e. pointers that point to other pointers. For example:

```
int *p;         // p is a pointer to an int, with an undefined initial value
int **pp = &p; // pp is a pointer to a pointer to an int and points to p
int i;          // i is an int, with an undefined initial value
*pp = &i;       // set p (the thing pointed to by pp) to point to i
*p = 23;        // set i (the thing pointed to by p) to 23
```

Needless to say, if applied to excess this can make your code harder to read.

Exercise 3.4: What is the output of the following program? Can you explain why? (Hint: A good way of explaining it might be to use a diagram.)

```
#include <iostream>

int main()
{
  int i = 23, j = 9;
  int *p = &i;
  int *q = &j;
  int **pp = &q;
  **pp *= 2;
  pp = &p;
  **pp -= j;
  std::cout << i << '\n';
  return 0;
}
```

References

In addition to pointers, C++ also allows us to express indirection using *references*. Conceptually, references are very similar to pointers, but they differ syntactically. Moreover, whilst it is possible to change the variable to which a (non-const) pointer points at any point during its lifetime, a reference must be initialised to refer to a variable, and then cannot be *reseated* (i.e. once it refers to a variable, it refers to it for all time).

Reference variables in C++ are those that have a reference type: syntactically, such types are written using an ampersand (`&`). For example, the type 'reference to int' would be written as `int &`. To initialise a reference, you just write the name of the variable to which you want it to refer. To modify the value of the referred-to variable, you just assign to the reference as if it were simply another name for the original variable. For example:

```
int i = 9;
int& r = i; // r refers to i
r = 23;     // set i to 23
```

As you can see, references can be significantly easier to use than pointers from a syntactic perspective.

3.6 Arrays

Working with individual values is all well and good, but there are often times when we need to be able to represent a *sequence* of values instead: for example, consider making a list of all of the students taking this course. C++ provides basic support for sequences in the form of *arrays* (these were inherited from C). An array is a fixed-size sequence of values, all of the same type, which are stored contiguously in memory. The size of an array must be a compile-time constant, and cannot change at runtime: this makes arrays simple, but somewhat inflexible. More flexible support for sequences is available via containers such as `std::vector` in the C++ Standard Library (see §3.13), but we briefly cover arrays here for completeness. Arrays can be declared using the following syntax:

```
int arr[3]; // declares an array of 3 ints
```

Like normal variables, arrays can also be initialised, in this case using the following syntax:

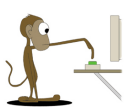
```
int arr[3] = { 23, 9, 84 }; // specifying the size (3) is optional here
```

The elements of an array are indexed from 0 upwards, and individual elements of an array can be accessed using their index in constant time. For example, the following could be used to set each element of `arr` to the square of its index in the array:

```
for(int i = 0; i < 3; ++i)
{
  arr[i] = i * i;
}
```

The expression `arr[i]` yields a reference to element `i` of `arr`.

Exercise 3.5: Write a program that initially creates an int array of size 10 and fills it with the first 10 powers of 2 (the first element of the array should be $2^0 = 1$). Your program should then print out the contents of the array in reverse order.



3.7 Functions

Up until now, we've been writing all of our code in the `main` function, but this is not an approach that will scale to larger programs. A more maintainable approach is to break our program into reusable pieces that have a well-defined purpose, and then make use of these to achieve our overall goal: one simple mechanism that C++ provides us with to achieve this is the notion of a *function*.

Functions in C++ are a bit like functions in mathematics, but with the crucial difference that they can have external side-effects (e.g. they can change the program state or output things to the terminal). Of course, not all functions *have* to change state: functions that don't are known as *pure* functions, and essentially model the function concept you are familiar with from mathematics. For example, recall the following definition of a factorial number:

$$\text{fac}(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times \text{fac}(n-1) & \text{otherwise.} \end{cases} \tag{3.1}$$

In C++, this can be written as:

```
1  int fac(int n)
2  {
3    if(n == 0) return 1;
4    else return n * fac(n-1);
5  }
```

The following example shows how `fac` can be used:

```
1  #include <iostream>
2
3  int fac(int n)
4  {
5    // As above
6  }
7
8  int main()
9  {
10   std::cout << fac(3) << '\n'; // prints 6
11   return 0;
12 }
```

More formally, functions in C++ can be defined using the syntax:

```
1  returnType functionName(parameter1, parameter2, ...) { statements }
```

In this, `returnType` specifies the type of the value returned by the function, `functionName` is the name of the function, each `parameter` consists of a type followed by a name for the parameter, and `statements` are the body of the function, which is enclosed in a block ({}).

Exercise 3.6: Write a function called `fib` that computes numbers $0, 1, 1, 2, 3, 5, \ldots$ from the Fibonacci sequence. Recall from mathematics that each term of this sequence (aside from the first two terms) can be computed as the sum of the two terms immediately preceding it. Your function should be such that `fib(0) = 0`, `fib(1) = 1` and `fib(4) = 3`. You need not worry about the efficiency of your implementation for the purposes of this exercise.

Side Effects

As mentioned above, functions in C++ do not have to be pure: they can have side effects. As a simple example of an impure function[1], consider the following:

[1] We have actually already seen examples of such functions in the form of the `main` function in most of the programs we have written thus far.

```
#include <iostream>

void foo()
{
  std::cout << "Foo!\n";
}

int main()
{
  foo();
  return 0;
}
```

Here, `foo` is not a pure function because it has a side effect: it outputs 'Foo!' to the terminal.

Modifying External Variables

Another example of a side effect would be changing the value of a variable that is not local to the function. To understand how this is possible, we need to understand what happens to a function's arguments when a call to it is made.

Consider the function call `fac(3)`: as we have seen above, conceptually this passes the value 3 to the function `fac`, which returns 6 as a result. However, on a practical level, what actually happens is that the program maintains a data structure known as the *call stack*, which contains a section for each invocation of a function that has not yet returned. When a call to a function (e.g. `fac`) is made, its arguments (e.g. 3) are pushed (copied) onto the call stack, from where they can later be retrieved by the called function.[2]

If the type of the parameter corresponding to an argument is not a reference type (e.g. it is a normal or a pointer type), then the value of the argument is pushed onto the stack (if the type is a pointer type, this means pushing an address); if it is a reference type, then the address of the argument is pushed onto the stack. In the former case, an argument may be referred to as being passed 'by value'; in the latter case, it may be referred to as being passed 'by reference'.

By passing a function either a pointer or a reference to a variable that is not local to the function, we can allow the function to change the value of that variable. For example:

```
#include <iostream>

void set_to_dob(int *p, int& r)
{
  *p = 23;
  r = 9;
}

int main()
{
  int x, y;
  set_to_dob(&x, y);
  std::cout << x << ' ' << y << '\n'; // prints 23 9
  return 0;
}
```

Note that people may also sometimes colloquially refer to `x` as having been passed by reference in this example: strictly speaking, what has happened is that a pointer to `x` has been passed by value, but the end result is the same as if `x` had indeed been passed by reference.

[2] The order in which the arguments are pushed depends on the *calling convention* being used: that's beyond the scope of this course.

Exercise 3.7: As shown in the following program, the standard C++ function `sqrt` computes and returns the square root of its argument:

```
#include <cmath>
#include <iostream>

int main()
{
  std::cout << sqrt(9.0) << '\n'; // prints 3
  return 0;
}
```

However, as implemented, it is only defined for non-negative inputs: if you call it on a negative number, you'll get what is known as a *domain error*. To make this behaviour slightly more friendly, your task in this exercise is to write a wrapper function, `mysqrt`, with the following behaviour:

```
/**
 * If n >= 0, sets failed to false and returns sqrt(n).
 * If n < 0, sets failed to true and returns 0.
 */
T1 mysqrt(double n, T2 failed);
```

What are appropriate types for T1 and T2?

3.8 Dynamic Memory Management

The values of the variables that we use in our program don't just exist in a vacuum: they have to be stored somewhere in the computer's memory. When we declare a variable that has local scope (see §3.4), its value will be stored on the call stack (specifically, it will be stored in the stack frame of its containing function). Such a variable will be destroyed when the frame containing it is removed from the stack (i.e. when the function returns).

Storing variables on the stack is convenient, but there are generally some variables in our program that would be better stored somewhere else, for several reasons. Firstly, we have to know the size of stack variables at compile time: this can make life difficult, e.g. if we want to store an unknown number of inputs from the user in an array. Secondly, we have limited control over the lifetime of such variables: when the function returns, they are automatically destroyed. (We can certainly return a variable's *value* and store it in another local variable in the calling function, but the memory for the original variable is still deallocated when its function returns.) Thirdly, the size of the call stack is generally limited to only a small fraction of the computer's overall memory: if we try to declare a huge fixed-size array, it's possible to cause a stack overflow.

As a result of these limitations, C++ provides us with an alternative: instead of storing every variable on the stack, we can store some variables in a separate block of memory known as the *free store*. Memory on the free store can be allocated and deallocated dynamically during the execution of the program, and the programmer has complete control over when allocation/deallocation happens. Moreover, there is generally much more memory available on the free store than on the call stack.

Allocation and deallocation of individual objects on the free store is performed using two built-in C++ operators: `new` and `delete`. The `new` operator allocates sufficient memory on the free store for an individual object of some specified type, constructs an object into that memory and returns a pointer to the object. The `delete` operator takes a pointer to an individual object that is stored in some memory on the free store, destroys the object and deallocates the memory. For example:

```
int *p = new int; // allocates a new int on the free store
*p = 23;          // sets the int pointed to by p to 23
delete p;         // deallocates the int pointed to by p
```

Every call to `new` must ultimately be matched by a corresponding call to `delete`, or what is known as a *memory leak* will result. In practice, this is easy to forget, and is one of the key downsides of such manual memory management. We will see a better alternative in §3.13.

In addition to allocating individual objects on the free store, it is also possible to allocate and deallocate memory for arrays of objects using two separate operators called `new[]` and `delete[]`. The size of such an array no longer needs to be a compile-time constant. For example:

```
int n = std::max(input_number(), 2);
int *arr = new int[n]; // allocate an array of n ints on the free store
arr[0] = 23;           // set elements of the array pointed to by arr
arr[1] = 9;
delete [] arr;         // deallocate the array pointed to by arr
```

As with individual objects, every call to `new[]` must ultimately be matched by a corresponding call to `delete[]`. Note that it is a heinous sin to call `delete` on a pointer allocated by `new[]`, or `delete[]` on a pointer allocated by `new`: if you do this, it results in undefined behaviour.

> **Question:** Think about how `new[]` and `delete[]` might work behind the scenes. What does `new[]` need to do to ensure that `delete[]` knows how much memory to deallocate?

3.9 Const Correctness

All of the variables we have seen in our programs so far have been modifiable: provided a variable is in scope, we have been able to change its value arbitrarily without any attempt by the compiler to stop us. However, in practice there are generally variables in our programs that we do not expect to change (for example, consider a variable representing the mathematical number π). Whilst it is certainly possible to simply rely on our abilities as programmers to avoid changing the wrong variables, people are fallible: it would be nice if we could tell the compiler which variables we think should remain constant, so that it can help us enforce this.

The mechanism C++ provides for this is the `const` keyword. By changing an object's type to make it `const`, we provide the compiler with the information it needs to issue an error message when we try and change the object and thereby alert us to a potential mistake. For example:

```
double d = 23.0;
d = 9.0;  // fine

const double PI = 3.141592654;
PI = 9.0; // compiler error!
```

As you can see, by declaring `PI` to be of type `const double` here, we have told the compiler that assignments to `PI` should be prevented.

We can also apply `const` to pointer and reference types. In the case of pointer types, there are two things that can be made `const`: either the pointer itself, or the object to which it points (the *pointee*). For example:

```
int i;
int *pi = &i;              // pi can change, and i can change via pi
const int *pci = &i;       // pci can change, but i cannot change via pci
```

```
int *const cpi = &i;         // cpi cannot change, but i can change via cpi
const int *const cpci = &i; // cpci cannot change, and i cannot change via cpci
```

Importantly, note that making the pointee `const` doesn't mean that you can't change the pointee at all, it just means that you can't change it *via that pointer*. As mentioned, it is also possible to apply `const` to reference types: this simply means that the object to which the reference refers (the *referee*) cannot be changed *via that reference*:

```
int i;
const int& r = i; // i cannot be changed via r
```

There is no way of specifying the `const`-ness of the actual reference: references are inherently `const` in C++, since they cannot be reseated to refer to a different object after being initialised.

Using the `const` keyword to annotate variables that should not change can be a powerful tool, but it is most powerful when applied consistently across a code-base. Programs that apply `const` in this consistent way are known as *const correct*. For more information on const correctness, interested students are advised to look at the following FAQ: https://isocpp.org/wiki/faq/const-correctness.

Exercise 3.8: A pointer to a non-const object of type T (i.e. a `T*`) can always be assigned to a pointer to a const object of type T (i.e. a `const T*`). As a result, the following code, which attempts to assign an `int**` to a `const int**`, might seem superficially plausible. In reality, however, it fails to compile. Can you explain why?

```
int main()
{
  int i;
  int *p = &i;
  int **pp = &p;
  const int **ppc = pp;
  return 0;
}
```

3.10 User-Defined Types

Whilst it is possible to write programs that only use primitive data types such as `int` and `double`, it would be extremely restrictive in practice, and would make the resulting code harder to read. Certain pieces of data, such as the components of a vector, naturally belong together: whilst we can in principle pass them around our program separately, it often makes much more sense to group them together into a bigger object and pass that around instead. The mechanism C++ provides to allow you to group pieces of data together is that of defining your own types.

There are various different ways of introducing a new type in C++, and we will not cover all of them in detail in this course. In Chapter 6, we will look at how to group both data and functions together into `class` types in the context of object-oriented programming. At this point in the course, however, it makes sense to start by looking at simple user-defined types that only contain data. By convention, such data types tend to be introduced using the C++ keyword `struct`, rather than `class`, although in practice there are only minor differences between `class` and `struct` types.[3] The following example shows how to use a simple `struct` type to represent a 2D vector:

[3] The differences are that access and inheritance are public by default for a `struct`, and private by default for a `class`, but we will elide over these details in this course.

```cpp
#include <cmath>
#include <iostream>

struct Vec2
{
  double x;
  double y;
}; // this semi-colon is important

Vec2 add(const Vec2& v1, const Vec2& v2)
{
  Vec2 result;
  result.x = v1.x + v2.x;
  result.y = v1.y + v2.y;
  return result;
}

int main()
{
  Vec2 v1, v2;
  v1.x = 3; v1.y = 4;
  v2.x = 5; v2.y = 6;
  Vec2 result = add(v1, v2);
  std::cout << result.x << ' ' << result.y << '\n'; // prints 8 10
  return 0;
}
```

To define the type, we use the `struct` keyword followed by the desired type name and a body (enclosed in curly braces) that contains the pieces of data that we want to group together (in this case, the two components of the vector). To access a piece of data within an instance of the type, we use the dot (`.`) operator: for example, to access the `x` component of `v1` in the above, we write `v1.x`. To access a piece of data within an instance of the type via a pointer to that instance, one option is to dereference the pointer and then use the dot notation just described:

```cpp
Vec2 v;
Vec2 *p = &v;
(*p).x = 23;
```

However, this operation is quite common, so C++ also provides a special syntax for it: the arrow (`->`) operator:

```cpp
p->x = 23; // means the same as (*p).x = 23;
```

The definition of a type is terminated with a semi-colon: this is needed because the grammar of C++ allows us to declare instances of the type alongside its definition. For example:

```cpp
struct S
{
  int i;
} s; // declares a variable s of type S
```

Having defined the type, we can create instances of it and pass them around our code. Observe that we are now able to return a 2D vector as a single object. Since functions can only have one return value, we did not previously have an easy way of writing a function to perform vector addition. Instead, we would have had to write something like the following, which is much less readable:

```cpp
void add(double x1, double y1, double x2, double y2, double& xr, double& yr)
{
```

```
  xr = x1 + x2;
  yr = y1 + y2;
}
```

One additional point that is worth making is about the way in which we should pass objects of user-defined types to functions. Recall from §3.7 that variables are passed to functions either by value or by reference, and that passing by value involves copying the variable onto the stack, whilst passing by reference involves copying its address onto the stack. This can now be seen to have important implications: objects of user-defined types may be large, making copying them onto the stack potentially costly. It is therefore generally preferable to pass objects of such types by `const` reference rather than by value.[4] An example of passing by `const` reference can be seen in the `add` function above.

Exercise 3.9: Try refactoring the following (very limited) example of complex numbers to use a `struct` type:

```
#include <iostream>

void mult(double r1, double i1, double r2, double i2, double& rr, double& ir)
{
  rr = r1*r2 - i1*i2;
  ir = i1*r2 + r1*i2;
}

int main()
{
  double r1 = 3, i1 = 4;
  double r2 = 5, i2 = 6;
  double rr, ir;
  mult(r1, i1, r2, i2, rr, ir);
  std::cout << rr << " + " << ir << "i\n";
  return 0;
}
```

Add functions to implement some other complex number operations, such as addition, subtraction, division, etc.

3.11 Templates

When we defined the `Vec2` type in §3.10, some of you may have wondered why the `x` and `y` components contained within it necessarily had to be of `double` type. A `double` is essentially C++'s model of a real number, so the `Vec2` type effectively models vectors over $\mathbb{R}^2$. It is easy to imagine wanting to model vectors over other 2D vector spaces such as $\mathbb{Z}^2$ or $\mathbb{C}^2$.

More generally, suppose we simply want to model pairs of objects of different types. For example, we might want both a `(std::string,int)` pair to model a student and their Bod card number, and a `(double,double)` pair to model a real number and its square. Our first thought might be to introduce different types for each of these possibilities:

```
struct StringIntPair
{
  std::string first;
  int second;
};

struct DoubleDoublePair
```

[4] In C++11 and above, the advent of *move semantics* has changed this picture somewhat. However, you will not go seriously wrong if you follow this advice for now.

```
{
  double first;
  double second;
};
```

However, this quickly gets extremely repetitive and boring. The structure of the two `struct` types in the above is exactly the same: the only difference between them is in the types they use for `first` and `second`.

Fortunately, C++ has a mechanism for dealing with this problem, known as *templates*. This involves the programmer defining a templated type that makes use of one or more type parameters that can be specified when the template is used. For example, the above two `struct` types can be replaced by the following `struct` template:

```
template <typename T1, typename T2>
struct Pair
{
  T1 first;
  T2 second;
};
```

Here, `T1` and `T2` are the type parameters, and `first` and `second` will be given different types based on what those parameters are specified to be. For example, to make a `(std::string, int)` pair, the programmer simply needs to *instantiate* the template type by specifying that `T1 = std::string` and `T2 = int`. This can be done as follows:

```
Pair<std::string,int> p;
p.first = "Foo";
p.second = 23;
```

Instantiating the template with different `T1` and `T2` types leads to different pair types in the resulting code. For example, `Pair<std::string,int>` and `Pair<double,double>` are completely different types that just happen to share a common template. Roughly speaking, the way in which templates work behind the scenes is that the compiler finds all of the places in which different instances of the template are used in the program, and makes new types by substituting in the various combinations of type parameters involved.

There is a lot more that could be said about templates, but unfortunately most of it is somewhat outside the scope of this course for time reasons. For more information, we suggest that interested students consult *C++ Templates: The Complete Guide* by David Vandevoorde and Nicolai M. Josuttis.

3.12 Error Detection and Handling

So far, we have been assuming that nothing ever goes wrong during the execution of a program, but in practice this is not the case. Problems can arise for all kinds of reasons, including programmer error, incorrect input provided by the user, and unexpected issues caused by the environment in which the code is being run (for example, other programs running on the machine could be using all of the available memory). Programs need to cope with such difficulties if we want them to be robust enough to deploy in a production environment. C++ provides a number of mechanisms that programmers can use for error detection and handling, and we briefly discuss each of them in turn in what follows.

Assertions

One simple mechanism a programmer can use to detect errors is to specify things that ought to be true at specific points in a program. For example, they might specify that the input to a square root function should be non-negative. In C++, this can be achieved using *assertions*. An assertion takes the following form:

```
assert(condition);
```

It specifies that `condition` ought to be true at the point at which program execution reaches the assertion. For example:

```
#include <cassert> // make the assert macro available

double mysqrt(double n)
{
  assert(n >= 0.0);
  //...
}
```

If assertion-checking is enabled, a failed assertion will cause the program to halt, allowing the cause of the problem can be identified.

Inline Error Handling

Whilst assertions can be useful, assertion-checking is often turned off in production systems for performance reasons.[5] This is potentially quite dangerous: it means that certain conditions that should arguably be being checked at runtime are not actually being checked at all in production code. For example, consider the following simple function that is intended to safely get an element of an array of length `len`:

```
int get(int *arr, int len, int i)
{
  assert(0 <= i && i < len);
  return arr[i];
}
```

With assertion-checking turned off, this is no safer than a function that just returns `arr[i]` without checking whether `i` is within the array bounds. A better alternative is to simply change the assertion into an `if` statement:

```
int get(int *arr, int len, int i)
{
  if(0 <= i && i < len) return arr[i];
  else return -1; // somewhat arbitrary
}
```

However, this then leaves us with the problem of what to do when the condition is not satisfied. Terminating the program (like an `assert` would do) seems draconian, but returning -1 seems unsatisfactory too: callers of `get` now have to explicitly check whether `get` returns -1 and handle that case appropriately; worse, if any of the array elements are themselves -1, callers will be unable to tell whether `get` is returning a valid array element or signalling an error:

```
int result = get(arr, len, i);
if(result == -1)
{
  // Now what?
}
```

One possible solution might be to change the signature of the function:

[5] One can debate whether or not this is a good idea: it has been likened to carrying a parachute when you're testing a plane, but leaving it behind when you go on a mission.

```
bool get(int *arr, int len, int i, int& result)
{
  if(0 <= i && i < len)
  {
    result = arr[i];
    return true;
  }
  else return false;
}

//...

int result;
if(get(arr, len, i, result))
{
  // Use result
}
```

This works, but feels clunky: you now have to declare a separate variable (`result`) in order to call `get`, whereas what you would like to be able to do is just pass the result of `get` around directly without worrying about potential errors.

Exceptions

Exceptions are designed to be a solution to problems such as this: their essential premise is that whilst problems inevitably arise during program execution, they only do so in exceptional circumstances. Most of the time, things will just work: it's undesirable to have to clutter up your code with lots of error-checking code, particularly when most of the time it's not actually being used. For example, this just looks ugly:

```
bool succeeded = tryThis();
if(!succeeded)
{
  //...
}

succeeded = tryThat();
if(!succeeded)
{
  //...
}

succeeded = tryTheOtherThing();
if(!succeeded)
{
  //...
}
```

It would be much nicer to just be able to write the following, and move all of the code to handle the failure cases elsewhere:

```
tryThis();
tryThat();
tryTheOtherThing();
```

This is exactly what C++'s exception mechanism allows us to do. In simple terms, it works as follows: the programmer surrounds code that can potentially fail in a `try` block, and provides `catch` handlers associated with that `try` block to handle exceptions (failures) of various different types that occur within the `try`. The code within the `try` can signal the occurrence of such a failure by `throw`-ing an exception: this causes execution to jump to a suitable `catch` handler, which can then deal with the problem appropriately. For example:

```
#include <iostream>
#include <stdexcept>

double mysqrt(double n)
{
  if(n < 0.0) throw std::runtime_error("The input to mysqrt must be positive");
  return sqrt(n);
}

int main()
{
  try
  {
    double n;
    std::cin >> n;
    std::cout << mysqrt(n) << '\n';
  }
  catch(std::exception& e) // a supertype of std::runtime_error
  {
    std::cerr << e.what() << '\n';
  }

  return 0;
}
```

Some of you may be wondering what happens when execution jumps from one function to another: for example, the exception here is thrown in `mysqrt`, but caught in `main`. The answer is that the call stack is unwound up to the first function in which the exception can be caught: here, execution leaves `mysqrt`, the stack frame for the current invocation of `mysqrt` is destroyed, and execution jumps to the exception handler in `main`, which prints an error message. This ability to `throw` an exception in one place and `catch` it in another is one of the most powerful features of the exception-handling mechanism: it allows us to separate error detection (knowing that an error occurs) from error handling (knowing what to do about it). This is important, because although low-level code such as `mysqrt` might be able to detect an error, it is rarely in a position to know what to do about it: without exceptions, we'd have to laboriously transfer information about the error back up to the higher-level code that can deal with it.

3.13 The C++ Standard Library

The Standard Library is a fundamental part of C++. It provides a comprehensive set of generic and efficiently-implemented classes that can be useful for most applications. These classes can be used with any built-in types and with any user-defined types that support some elementary operations such as copying and assignment. Modern compilers are very well optimized to minimize any (abstraction) penalty that could arise from the heavy use of the Standard Library. The Standard Library includes many useful classes: we will focus on the basics – containers and iterators, streams, algorithms and smart pointers. Note that there are multiple versions of the Standard Library (as with C++ itself). If you can use a modern compiler and backwards compatibility is not a concern, use C++11/C++14, as this version offers much more, including smart pointers. A good reference for the Standard Library can be found on this webpage: http://en.cppreference.com.

Containers and Iterators

Containers and associative containers will be widely used on a daily basis in whatever application you decide to implement. In the following, we will only cover a few of them. Refer to http://en.cppreference.com/w/cpp/container for more details.

Vectors We start with `std::vector`, on which we will demonstrate the basic principles that apply to other parts of the Standard Library as well. A `std::vector` is a sequence container that represents a resizable array. A `std::vector` automatically extends its size to make space for any elements we push into it. This is in sharp contrast to raw arrays (see §3.6). For example, the following code creates an empty vector of `double`s and then appends two elements to it. Note that we do not need to worry about making enough space for these elements: `vector` will take care of this behind the scenes.

```
#include <vector> // include vector class template

int main()
{
  std::vector<double> my_vct;
  my_vct.push_back(3.14);
  my_vct.push_back(1.0);
  return 0;
}
```

Note that entities from the Standard Library are preceded with `std::`, which allows them to be distinguished from other entities with the same name. In addition to creating `vectors` that are initially empty, we can also directly create a `vector` with a particular size. For example:

```
std::vector<int> my_vct(10); // vector with 10 ints (all initially 0)
```

If no initial value is specified for the elements of the vector, they are initialised with the default value for their data type. Alternatively, an initial value can be specified by passing a second argument when constructing the `vector`:

```
std::vector<int> my_vct(10, 3); // vector with 10 ints (all initially 3)
```

As with raw arrays, elements in `std::vector` are 0-indexed and can be accessed using square brackets notation. For example:

```
my_vct[0] = 0;  // change 1st element to 0
my_vct[5] = 7;  // change 6th element to 7
```

The `std::vector` interface provides many other useful methods such as `size()`, `clear()`, `pop_back()`, etc. Refer to http://en.cppreference.com/w/cpp/container/vector for details.

Exercise 3.10: Allocate a `vector` of 5 integers initialized to 1, and add the following numbers to it: $\{5, 3, 10\}$. Print out the size of the vector and all of its elements. Next, modify the last element to 8 and print out all elements again. Finally, erase all elements from the vector, and print the size of the `vector`.

Iterators The Standard Library provides many classes that internally implement very different and often not much compatible data structures. However, it would be very convenient to have a common access to the data structures; otherwise, if we decided to change type of the container, we would need to re-implement lot of code. All containers allow to allocate

- `iterator` – iterates over the elements from the start position one by one. Elements can be changed.

- `const_iterator` – iterates over the elements from the start position one by one. Elements cannot be changed.

and implements the following iterators:

- `begin/cbegin` – returns an iterator / `const` iterator to the beginning
- `end/cend` – returns an iterator / `const` iterator to the end

Typically, we use iterators with `for` loops

```
1 std::vector<int> m_vct(10, 5); // vector of 10 elements initialized to 5
2 for(std::vector<int>::const_iterator it = m_vct.cbegin(); it != m_vct.cend();
    ↪ ++it)
3 {
4     std::cout << " " << *it; // print each element
5 }
```

Exercise 3.11: Allocate a vector of 15 `double`s. Implement a `for` loop in which each element will be replaced by distance of the current element to the first one.

The `std::vector` is (perhaps) the most widely used container, however Standard Library provides many more:

Containers

- `vector` – sequence container that implements storage of data elements. You can think about it as a very convenient dynamically allocated array into which we are allowed to add more elements on-the-fly. Similarly, we can also remove elements. Importantly, the elements are stored contiguously, which means that elements can be accessed not only through iterators, but also using offsets on regular pointers to elements.
- `list` – is a container that supports constant time insertion and removal of elements from anywhere in the container.
- `deque` – is an indexed sequence container that allows fast insertion and deletion at both its beginning and its end. As opposed to `vector`, the elements of a `deque` are not stored contiguously: typical implementations use a sequence of individually allocated fixed-size arrays. In addition, insertion and deletion at either end of a deque never invalidates pointers or references to the rest of the elements.

Associative Containers

- `set` – is an associative container that contains a sorted set of unique objects
- `map` – is a sorted associative container that contains key-value pairs with unique keys

Note, that the Standard Library contains also unordered versions of associative containers, `LIFO`, `FIFO` data structures and many more. Refer to details at http://en.cppreference.com/w/cpp/container

Exercise 3.12: Allocate a `set` container for integers, insert the following numbers $\{1, 5, 3, 10\}$ and print out the size of the set. Next, insert $\{5, 3, 8\}$ and print all elements out. Print also the number of elements in the set. Finally, allocate an `std::vector`, insert the same sequence of numbers, print all elements out and compare the outputs.

Question: What are the differences between `std::vector` and `std::set`?

Exercise 3.13: Think about potential use cases for other (associative) containers.

Stream I/O

Often, we need to send to or receive information from console, file, or any other device. In general, this is a very difficult task. Here, the Standard Library comes to rescue again. In fact, it is one of the most powerful and flexible tools that Standard Library offers as it is very straightforward to write input and output routines for any application.

Without delving deep into the details, *streams* are internally series of characters (`char` or `wchar_t`). Any object that can be written to one stream, can be written to all types of streams. In other words, you can read/write data to/from console, file or any other media with the same interface without the need to know the details how to write to the storage media. Streams work with all build-in types; and also with user-defined types. The only requirement is that objects need to implement insertion operator << that puts object into streams, and the extraction operator >> that reads objects from stream.

In the previous chapters, we have often used `std:cout` object and insertion operator << to print data to the console:

```
#include <iostream> // input/output stream
std::cout << "Hello world" << std::endl; // print to console and flush the
  ↪  output buffer
```

Similarly, we can also read data from the standard input (keyboard) with `std::cin` object and extraction operator >>

```
std::string my_string;
std::cin >> my_string;
std::cout << "User provided: " << my_string << std::endl;
```

We have been using streams for reading and writing to `console` extensively in the previous chapters. Now, we will try to write text into file. To this end, we will use `std::ofstream` class from `<fstream>` header. Refer to documentation at http://en.cppreference.com/w/cpp/io/basic_fstream

```
std::ofstream fout("filename.txt"); // create new file

if(!fout.is_open()) // check whether file exists
  throw std::runtime_error("Can't open file");

fout << "Hello world\n";

fout.close(); // close stream (not necessary)
```

Up to this point, we have been using only the build-in types that support the extraction (>>) and insertion (<<) operators by default. Can we use stream with user-defined objects as well? Yes, we just need to provide the respective operators

```
#include <iostream>

struct Parameters
{
  int parameter_a;
  double parameter_b;
```

```
  std::string name;
};

std::ostream& operator<<(std::ostream& os, const Parameters &params)
{
  os << "Name: " << params.name << "\n";
  os << "Parameter A (int): " << params.parameter_a << "\n";
  os << "Parameter B (double): " << params.parameter_b << "\n";

  return os;
}

int main()
{
  Parameters m_params;
  m_params.parameter_a = 5;
  m_params.parameter_b = 3.14;
  m_params.name = "My random parameters";

  std::cout << m_params << std::endl;

  return 0;
}
```

Exercise 3.14: Implement a program that asks user about her name and age, store it into an appropriate structure and write it out into a text file (use operator <<).

More information about streams can be found at http://en.cppreference.com/w/cpp/io

Smart Pointers

One of the most powerful feature of C++ are pointers. As we have already seen, they enable a very efficient memory management. However, this efficiency comes at a cost – while most bugs are usually detected by compiler, pointer misuse typically needs to be discovered by programmers. In fact, pointer misuse is perhaps the most common and major source of bugs in C++. A classic example is a memory leak

```
for(int i = 0; i < 10000; ++i)
{
  int *p = new int[1024];
  // do something with *p
}
```

Will it compile? Yes. Will it run? Yes. So what is wrong with this code excerpt and how can we find this bug? The problem is, that we allocate an array of 1024 integers in each iteration of this array but we never deallocate this memory (remember, every `new` must be matched by `delete`[6]). In other words, we have allocated $10000 \times 1024 \times$ `sizeof(int)` bytes of unused memory. Finding such bugs, despite we can detect memory leaks in Visual Studio and we have tools such as `valgrind`, is often very, very time consuming process.

Since the C++11 standard the Standard Library contains *smart pointers* which are intended to reduce bugs caused by the misuse of pointers. Smart pointer is a class template that simulates a pointer while providing features that enable automatic, exception-safe, object lifetime management. We will demonstrate this on *shared pointer* (other options are `std::unique_ptr` and `std::weak_ptr`).

[6] A careful reader might ask why we have not faced such issues with `std::vector` and other containers. This is one of the advantages of using Standard Library – the memory management is efficiently handled by the containers itself (which typically deallocate all allocated memory in their destructors).

Shared pointer (`std::shared_ptr`) is a smart pointer that retains shared ownership of an object through a pointer. Several shared pointers may own the same object; the shared pointers count references to the object. The object is destroyed and its memory deallocated when either of the following happens:

- the last remaining `std::shared_ptr` owning the object is destroyed;
- the last remaining `std::shared_ptr` owning the object is assigned another pointer via operator= or reset().

In order to demonstrate this, we create an instance of the `Parameters` structure from the previous section and pass it to a smart pointer

```
#include <iostream>
#include <memory>     // header for smart pointers

struct Parameters
{
  int parameter_a;
  double parameter_b;
  std::string name;
};

int main()
{
    std::shared_ptr<Parameters> sp = std::make_shared<Parameters>();

    return 0;
} // all good, the allocated object is automatically deallocated
```

We can access member of `Parameters` with the standard operator `->`

```
std::shared_ptr<Parameters> sp = std::make_shared<Parameters>();

sp->parameter_a = 15;
sp->parameter_b = 1.7565;
sp->name = "Random parameters";
```

Similarly to raw pointers, we can specify `const`-ness

```
std::shared_ptr<Parameters> spi = std::make_shared<Parameters>(); // normal
 ↪  shared_ptr
std::shared_ptr<const Parameters> spci = spi; // spci can change but object
 ↪  cannot change via spci
const std::shared_ptr<Parameters> cspi = spi; // cspi cannnot change but object
 ↪  can change via cspi
const std::shared_ptr<const Parameters> cspci = spi; // cspci cannot change and
 ↪  object cannot change via cspci
```

We can also pass smart pointers to functions

```
#include <iostream>
#include <memory>     // header for smart pointers

struct Parameters
{
  int parameter_a;
  double parameter_b;
  std::string name;
};

```

```
11 void modifyParamA(const std::shared_ptr<Parameters> &cptr)
12 {
13     cptr->parameter_a = 345;
14 }
15
16 int main()
17 {
18     std::shared_ptr<Parameters> sp = std::make_shared<Parameters>();
19     modifyParamA(sp);
20     std::cout << sp->parameter_a << std::endl;
21
22     return 0;
23 }
```

Question: Shall I always use smart pointers instead of their raw counterparts?

In modern C++, the raw pointers should be used only in very small blocks of limited scope where performance is critical and there is no chance of confusion about ownership. Everywhere else should be used smart pointers. A rule of thumb: if you think your application is slow, it is usually much better to focus on algorithmic part as overhead of smart pointers is very limited.

Exercise 3.15: Implement a program that creates 3 shared pointers to some object, pass them into a function that modifies the object. Finally, print the current state of the object.

Further reading The Standard Library is a large library providing many useful data structures and algorithms. We have very briefly described only the most basic parts but there are many more. For instance, `algorithm` provides sorting, `chrono` accurate timers, `thread` / `mutex` build-in support for multi-threading, etc. For further reading, we suggest http://en.cppreference.com/w/cpp and Effective STL by Scott Meyers.

Chapter 4

Development Tools

4.1 Introduction

The previous chapters have shown you how to set up your compilers on Windows and Linux, and how to write and compile basic C++ programs with them, but in the process of doing this you may have encountered a few minor frustrations. In particular, you will probably either have written the same code out twice, once on each platform, or you will have had to manually copy and paste it from one platform to the other. Even more frustratingly, because Visual C++ and g++ have different ways of specifying which files to compile, you will have ended up having to set up the same project separately on both systems. This is not only time-consuming, it's also error-prone: you can easily end up with inconsistent builds if you change the build on one platform but not another.

In this chapter, we show how these irritations can be avoided by using proper development tools. Version-control systems such as Git make it easier to manage code that is being developed on multiple systems, and by multiple people, over a period of time. Cross-platform build systems such as CMake can be used to generate things like Visual C++ solutions or g++ Makefiles from a common build script, meaning that you can specify the files in a project in the same way on different platforms.

4.2 Version Control with Git

Version-control systems (VCS) are tools that help track and manage changes to files over time. They keep track of changes by means of a special database that records various aspects such as the time when something was changed, the difference between the earlier version and the current one (a patch), and comments from the person who made the changes about the changes themselves, among other things. This allows for one to turn back the clock, to recover from an error, or analyse comparisons to earlier versions to help debug issues. The virtues of version control also extend to serving as a record of the *process* of building the system or code-base itself.

Typically, programmers, particularly those working in teams, are continually adding or making edits to existing files. The code for a project is typically organized in a hierarchical structure of directories and files. Here, one person might be working on a new feature while another changes a file, say to fix a bug; each developer may make their changes in several parts of the file hierarchy. Version control helps deal with, and often avoid, these kinds of problems by tracking changes by each contributor and helping prevent concurrent work from conflicting. When there are conflicting issues to be resolved, VCS provide the means to solve them in an orderly manner, without breaking things for others.

If you're someone who hasn't used version control before, your approach to dealing with changes and collaboration may have often involved having different versions of your files (maybe with suffixes like `v1`, `new`, or `latest`), or even chunks of your file internals commented out or blocked out to disable different functionality. Version-control systems provide a better, more coherent abstraction for dealing with such issues. Ultimately, version control is an essential part of a programmer's armoury, and those who learn to use it effectively come to recognise its incredible value, and reap its benefits many times over.

In this course, we will familiarise ourselves with one particular version control system: `Git`. `Git` is a *distributed* VCS, which just means that the database (referred to above) in which it keeps the requisite information need not be in some single central location. The following are a brief introduction to `git` commands: just enough to get you going for the purposes of this course. However, you are *strongly* encouraged to explore the many features and facets of `git` by going through some of the many tutorials available online. In fact, for this particular section, we will engage in an interactive exercise exploring the basics of `git` with the help of the demo at https://try.github.io.

`git init`
: creates a new repository in your current directory, if one does not already exist.

`git clone [username@host:]/path/to/repository`
: create a working copy of a repository.

`git diff`
: view `diff` of your current changes to the working copy.

`git add <filename>`
: propose additions and changes, by adding to the *index*.

`git commit -m "Commit message"`
: commit changes to your copy of the repository, with the specified commit message.

`git log`
: view the repository history of commits

`git push origin master`
: send your committed changes to the remote repository; specifically, the master branch.

`git remote add origin <server>`
: connect an existing repository with a remote server

`git checkout myshinybranch`
: switch to another branch

`git checkout -b myotherbranch`
: create a new branch and switch to it

`git branch -d myotherbranch`
: delete a branch

`git pull`
: make your local repository *fetch* and *merge* remote changes.

`git merge <branch>`
: merge another branch into your active (current) branch.

`git diff <source_branch> <target_branch>`
: view `diff` between different branches

4.3 Cross-Platform Building with CMake

Overview

CMake is a cross-platform build tool. The easiest way to think of it is as a program that converts build scripts written in a platform-neutral way into platform-specific project files. In other words, you can specify how your project should be built in a single place, and then let CMake use this specification to generate a Visual C++ solution on Windows, or a Makefile, or an Xcode project on Mac OS X. This is a huge win, because it means that you don't have to worry about keeping your build files in sync across multiple platforms.

Downloading and Installing CMake on Windows

To download and install CMake on Windows, simply go to https://cmake.org/download/, download the installer for the latest release and run it to completion. If asked whether to add CMake to your system path, choose to do so. The installer will put the executables for CMake in `C:/Program Files (x86)/CMake/bin`: for the purposes of this course, the one you'll be using will be `cmake-gui.exe`.

Setting Up a Simple CMake Project

The first step in setting up a new CMake project is to create a directory for it and add a few files, initially empty.

Exercise 4.1: Go to `Documents/Visual Studio 2013/Projects` and create a new directory within it called `cmaketest`. Add new files called `main.cpp`, `Foo.cpp` and `Foo.h`.

CMake builds are specified in files called `CMakeLists.txt`. Add a `CMakeLists.txt` file to the `cmaketest` folder, and copy the script shown below into it.

```
1  cmake_minimum_required(VERSION 2.8)
2
3  PROJECT(cmaketest)
4
5  ############################
6  # Specify the target name #
7  ############################
8
9  SET(targetname cmaketestapp)
10
11 ############################
12 # Specify the project files #
13 ############################
14
15 SET(sources main.cpp Foo.cpp)
16 SET(headers Foo.h)
17
18 ############################
19 # Specify the source groups #
20 ############################
21
22 SOURCE_GROUP(sources FILES ${sources})
23 SOURCE_GROUP(headers FILES ${headers})
24
25 ######################
26 # Specify the target #
27 ######################
28
29 ADD_EXECUTABLE(${targetname} ${sources} ${headers})
```

To understand what this script does, let's go through it line-by-line. Line 1 specifies the minimum version of CMake that is required to generate projects from this script. Line 3 specifies the name of the CMake project: when CMake is used to generate a Visual C++ solution, this will be the name given to the solution itself. Line 9 creates a user-defined variable called `$targetname` and sets it to 'cmaketestapp'. This is the name we want to give the target (application or library) we're currently defining. Each target will become an individual project in a Visual C++ solution. We put it in a variable to avoid hard-coding the string 'cmaketestapp' everywhere (this makes it easier to change later). Lines 15 and 16 create user-defined variables that specify the source and header files for the target. Lines 22 and 23 specify that the source files and header files should respectively be grouped under folders called 'sources' and 'headers' in the Visual C++ IDE. Line 29 specifies that the target should be an application (an executable) with the specified name and project files.

Having specified the setup of the project using this script, the next step is to use CMake to generate a platform-specific project file from it: in our case, a Visual C++ solution.

Exercise 4.2: From `C:/Program Files (x86)/CMake/bin`, run the `cmake-gui.exe` executable. This will bring up the CMake GUI application that you will use to generate the Visual C++ solution. At the top of the window, you will see two text boxes, labelled 'Where is the source code:' and 'Where to build the binaries:'. Into these, you should enter (respectively) the absolute path to your `cmaketest` directory, and the absolute path to a subdirectory of it called `build` (note that this does not yet exist: you will be prompted to create it in a moment). In the bottom-left of the window, you will see two buttons, Configure and Generate (only Configure will initially be available). Press Configure, opt to create the `build` directory and accept the default choice of generator for the project (Visual Studio 12 2013 Win64) by clicking Finish. A configuration process will now run, at the end of which the Generate button will become available. Click it, and then open the Visual C++ solution `cmaketest.sln`, which you will find in the newly-created `build` directory.

If you look at the contents of the solution, you will see that it contains a project called `cmaketestapp`, which corresponds to the target name you specified in the `CMakeLists.txt` file. Within this project, you will find the project files you specified. You can now simply edit these files and build your program as you would if you weren't using CMake. The only time you need to go back to the CMake script is if you want to change the structure of your project, e.g. by adding or removing project files.

To run your program, you will need to make sure that you are trying to run the correct project (in this case, `cmaketestapp`). To do this, right-click on the `cmaketestapp` project in the Solution Explorer, and click 'Set As Startup Project'.

4.4 Further Reading

A much more detailed introduction to Git, together with illuminating explanations of the Git model and a thorough coverage of Git's many features, can be found in the book *Version Control with Git*, by Jon Loeliger and Matthew McCullough. We thoroughly recommend this book to interested students who want to understand how Git works 'under the hood'.

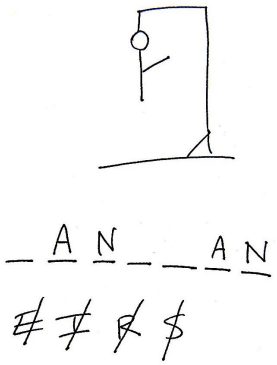
For CMake, there's a book called *Mastering CMake*, by Ken Martin and Bill Hoffman. Personally, we've never needed it, but it looks like it covers CMake fairly comprehensively if you're interested. As an alternative, there are tutorials and examples available on the web, and the CMake website has pretty good reference documentation.

Chapter 5

Mini-Project 1: Text Hangman

5.1 Game Overview

Hangman is a traditional word-guessing game for two players in which one player (Player 1) tries to guess a hidden word chosen by the other player (Player 2) by suggesting letters to fill in one at a time. Player 2 initially presents the hidden word to Player 1 as a sequence of underscores of the correct length. Correct guesses by Player 1 (those that correspond to a letter that is in the word) result in the sequence being updated by filling in the guessed letter in the correct position(s). Incorrect guesses result in Player 2 adding elements to a diagram showing Player 1 being progressively hanged from a scaffold (it is from this that the game gets it name). Consider the following example:



If Player 1 guesses the word before Player 2 completes the diagram, he wins; if not, he loses.

As another example, consider the first step of a game in which Player 2 chooses the hidden word to be 'food'. Player 1 is thus initially presented with a sequence of four underscores and must guess at a letter to fill in. If he chooses 'd', 'f' or 'o', the sequence is updated appropriately, e.g. choosing 'o' would result in the sequence '_oo_'. If he chooses any other letter, Player 2 will add his head to the diagram.

The goal of this mini-project is to implement a computerised variant of this game called *Text Hangman*, in which the computer takes the role of Player 2 in choosing hidden words and updating the hangman diagram.

5.2 Getting Started

The first steps are to clone the git repository and build the skeleton code.

Exercise 5.1: Clone and build the code using the following commands:

```
$ git clone https://bitbucket.org/gpcwm/texthangman.git
$ cd texthangman
$ ./build-win.sh 12 Debug
```

When you run the `build-win.sh` script for the first time, you may find that the directory containing `msbuild` (the tool needed to build Visual C++ solutions) is not on your system path:

to get the build working, you will need to modify the system path appropriately. Each version of Visual C++ has a different version of `msbuild` associated with it. In this case, we are using Visual Studio 2013, so you need to add `C:/Program Files (x86)/MSBuild/12.0/Bin` to the system path. If you are not sure how to do this, ask one of the demonstrators. Having added the `msbuild` directory to the system path, you should now be able to run the `build-win.sh` script successfully.

Having done this, open up the Visual Studio solution file (`texthangman.sln`) in `texthangman/build` and study the code. This solution is straightforward: all of the code is within a single file called `main.cpp` in a project called `texthangman`. The initial state of the code is shown in Figure 5.1.

5.3 Basic Implementation

If you run your version of the *Text Hangman* code (`Ctrl + F5` by default in Visual Studio), you will see that it doesn't currently do very much: it will just say 'Game Over!' and print the target word. To make it do something more interesting, you will need to implement the main game loop (at the place in the code indicated by one of the `TODO` comments). Each iteration of this loop should do a number of things:

1. It should render the hangman figure for the current score.
2. It should output the player's current knowledge of the target word.
3. It should prompt the player to guess a letter in the target word.
4. It should update the player's current knowledge of the target word and score based on this guess.

The loop should terminate when the player either wins or loses.

Exercise 5.2: Implement the main game loop as just described. What is an appropriate termination condition for the loop?

5.4 Rendering the Hangman Figures

The next task is to render the hangman figures using ASCII art. Corresponding to the initial score in the previous section, the hangman consists of 7 parts: head, torso, left arm, right arm, pelvis, left leg and right leg. One of these is added to the figure after each mistake by the player, in the order given. The final hangman figure (shown if the score reaches 0 and the player thereby loses the game) looks as follows:

```
  ------
  |    |
  o    |
 /|\   |
  ^    |
 / \   |
       |
---------
```

The hangman figures for higher scores look the same, but with fewer parts added. For example, the hangman figure for a score of 4 looks as follows:

```cpp
#include <iostream>
#include <stdexcept>  // for std::runtime_error
#include <string>

/**
 * \brief Generates a target word for a game of Text Hangman.
 *
 * \return  The target word.
 */
std::string generate_target_word()
{
  // TODO: Return a random target word from the words.txt file.
  return "wibble";
}

/**
 * \brief Renders the hangman figure associated with the specified score.
 *
 * \param score A score in a game of Text Hangman.
 */
void render_hangman(int score)
{
  // TODO: Render the hangman figure here.
  if(score > 0) std::cout << "Score: " << score << "\n\n";
  else std::cout << "Game Over!\n\n";
}

int main()
try
{
  std::string target = generate_target_word();
  std::string current(target.size(), '_');
  int score = 7;

  // TODO: Implement the main game loop.

  // At the end of the game, let the player know the final result.
  if(current == target)
  {
    std::cout << "Congratulations!\n";
  }
  else
  {
    render_hangman(0);
    std::cout << "Target Word: " << target << '\n';
  }

  return 0;
}
catch(std::exception& e)
{
  std::cerr << e.what() << '\n';
}
```

Figure 5.1: The initial state of the *Text Hangman* code.

```
  ------
  |    |
  o    |
 /|    |
       |
       |
       |
---------
```

These figures can be rendered in a variety of ways. One simple method might be to switch on the current score, but the resulting code is likely to be quite long-winded. Can you think of a better approach? (*Hint:* Instead of outputting the figures on-the-fly, consider constructing an explicit representation of it and updating that based on the current score.)

Exercise 5.3: Implement the rendering of the hangman figures. To do this, you should fill in the implementation of the `render_hangman` function, replacing the placeholder comment in the existing code.

5.5 Adding a Misses Set

As things stand at the moment, it is possible for the player to enter the same incorrect guess multiple times, decreasing their score each time. There is a reasonable argument to be made that this is unfair: the player is being punished multiple times for the same mistake.

A simple way to fix this problem is to add a `std::set` to the program that records incorrect guesses ('misses') and then ignores any incorrect guess made by the player if it has been seen before. To provide additional feedback to the player, it is also helpful to output this set in each iteration of the loop to remind the player that they do not need to try certain letters again.

Exercise 5.4: Implement a 'misses' set as just described.

5.6 Generating Random Words

In the initial code, the `generate_target_word` function always generates the same target word ('wibble'), making the game exceptionally boring if it is played repeatedly. To fix this, the game needs to generate a random target word each time it is played.

Exercise 5.5: If you look in the `texthangman` directory, you will see a file called `words.txt`, which contains a dictionary of English words, one per line. By replacing the `TODO` comment in the `generate_target_word` function, show how it is possible to randomly select a word from this file and use it as the target word for hangman. Note that we have set up the CMake script to copy the `words.txt` file from the `texthangman` directory into the `build` directory (where the executable resides) whenever the program is built, so you should be able to just open `words.txt` directly without having to specify an absolute or relative path.

Chapter 6

Object-Oriented Programming

6.1 Introduction

The style of programming (also known as a programming *paradigm*) you have been using so far in this course is known as *procedural programming*: you have been organising your code into functions (or procedures) that operate on pieces of data, but any connection between the functions and the data has been implicit rather than explicitly expressed in the code. In this chapter, we will look at another way of writing programs, known as *object-oriented programming*, in which functions are explicitly grouped together with the data on which they operate.

6.2 Classes

Object-oriented programming is based around two important concepts: *classes* and *objects*. Classes model concepts in the world, such as `Student`, or `Dog`. As such, the names they are given should generally be nouns. Objects are individual instances of classes, such as a student called Bob, or a dog called Fido. In code, classes can be used to group related functions and data together. For example:

```
1  class Dog
2  {
3  private:
4    std::string name;
5
6  public:
7    Dog(const std::string& name_)
8    : name(name_)
9    {}
10
11   void speak() const
12   {
13     std::cout << "Woof! My name is " << name << '\n';
14   }
15 };
```

The functions in a class are known as *member functions* to distinguish them from the global functions we have been looking at thus far. Objects of a particular class are constructed by invoking special member functions within the class, called *constructors*. For example:

```
1  // Calls Dog::Dog(const std::string&), passing in "Rover" as the name.
2  Dog rover("Rover");
```

Exercise 6.1: Implement a class called `RandomPatternGenerator` that can generate random grid-shaped patterns of a specified width and height whose elements are taken from a set of specified symbols. The following program shows the desired behaviour of your class:

```
// Add your code here
// ...
int main()
{
  int width = 5, height = 3;
  RandomPatternGenerator generator(width, height, "abcde");
  std::cout << generator.generate_pattern();
  /*
  Example Random Output:
  baddc
  cdabe
  edaab
  */
  return 0;
}
```

6.3 Inheritance and Polymorphism

In practice, many concepts in the world are not standalone: they can be seen to be subcategories of more general concepts. For example, both cats and dogs are kinds of animal. In object-oriented programming, we express this 'is a kind of' relationship between concepts using (public) *inheritance*, written as follows:

```
class Animal
{
private:
  std::string name;

public:
  Animal(const std::string& name_)
  : name(name_)
  {}

  virtual ~Animal() {} // for discussion

  void speak() const
  {
    std::cout << "My name is " << name << '\n';
  }
};

class Cat : public Animal
{
public:
  Cat(const std::string& name_)
  : Animal(name_)
  {}
};

class Dog : public Animal
{
public:
  Dog(const std::string& name_)
  : Animal(name_)
  {}
};
```

Derived classes inherit member functions (and variables) from their base class(es), so it's possible to call `speak` on a `Cat` or a `Dog`. Moreover, we can use a pointer/reference to an

instance of a base class to point/refer to an instance of one of its derived classes, and call speak via the base class pointer/reference:

```
Animal *p = new Dog("Fido");
p->speak();

Dog d("Necator");
Animal& r = d;
r.speak();
```

However, as currently written, this will just produce the speak behaviour associated with a generic animal: put bluntly, cats will not miaow, and dogs will not woof.

To fix this problem, C++ allows us to declare member functions as *virtual* in a base class, allowing them to be *overridden* with more specific behaviour in a derived class:

```
class Animal
{
  //...

  virtual void speak() const
  {
    // As before
  }

  //...
};

class Dog : public Animal
{
  //...

  // The virtual keyword here is optional, but makes your code more readable.
  virtual void speak() const
  {
    std::cout << "Woof! ";
    Animal::speak(); // call the base class's speak function
  }

  //...
};
```

Now, calling speak on an instance of `Dog` via an `Animal` pointer/reference will produce the desired behaviour:

```
Animal *a = new Dog("Rover");
a->speak(); // woofs as expected
```

How does this work behind the scenes? Well, in this situation, we can observe that `*a` really has two types: a compile-time (or *static*) type of `Animal`, and a runtime (or *dynamic*) type of `Dog`. When the compiler sees you trying to call a virtual member function such as `speak`, it performs what is known as *dynamic dispatch* to find the correct member function to call based on the dynamic type of the *receiving object* (which in this case is `*a`).

Virtual member functions and overriding are C++'s implementation of a computer science concept known as (runtime) *polymorphism*, the provision of a common interface to entities of different types. Providing a common interface in this way can be extremely useful. For example, suppose that instead of having a single animal, you have many animals of various different types, and you want them all to speak in their own particular way. In C++, this situation could be modelled by having a container of `Animal` pointers, and calling `speak` on each element of the container:

```
std::vector<Animal*> animals;
animals.push_back(new Cat);
animals.push_back(new Dog);
//...
for(size_t i = 0, size = animals.size(); i < size; ++i)
{
  animals[i]->speak();
}
```

Since `speak` is now virtual, each call will be dynamically-dispatched to the right function in one of the derived classes.

Exercise 6.2: Add a new base class called `PatternGenerator` and then have the class you wrote previously, `RandomPatternGenerator`, inherit from it. Then implement another derived class, `StripedPatternGenerator`, which generates a striped pattern instead. The following program shows the desired behaviour:

```
// Add your code here
// ...
int main()
{
  int width = 5, height = 3;
  std::vector<PatternGenerator*> patterns;
  patterns.push_back(new RandomPatternGenerator(width, height, "abcde"));
  patterns.push_back(new StripedPatternGenerator(width, height, "abcde"));
  for(size_t i = 0, size = patterns.size(); i < size; ++i)
  {
    std::cout << patterns[i]->generate_pattern() << '\n';
  }
  /*
  Exampled Striped Output:
  aaaaa
  bbbbb
  ccccc
  */

  return 0;
}
```

Are there any other interesting patterns that you can generate?

6.4 Further Reading

The notes in this chapter have only given you the very basics of what you need to write object-oriented programs in C++, so in practice you'd be well-advised to do some background reading. The course textbook, C++ Primer, contains lengthy chapters on Classes and Object-Oriented Programming, and we direct you in the first place to those. We also recommend that you take a look at *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma et al.

Chapter 7

Using Third-Party Libraries

7.1 Introduction

The programs we have been writing up to this point have been entirely self-contained, with their only dependency being the C++ Standard Library. However, the C++ Standard Library cannot, and explicitly does not try to, provide every bit of functionality that we could possibly need. For other functionality that we do not want to write ourselves, we can make use of the vast plethora of third-party libraries for all kinds of tasks that can be found on the web. In this chapter, we show how to download and make use of third-party libraries in our code. We use the AngelScript scripting library as an example: this is a library designed to allow C++ programs (especially games) to extend their functionality through external scripts. Understanding how to use third-party libraries will be important throughout the rest of this course, as we explore graphics programming using SDL in Chapter 8 and build a graphical game called *Nibbles* in Chapter 9.

7.2 The C++ Compilation Model

Before we look at the practicalities of using third-party libraries in our code, it makes sense to briefly explain the C++ *compilation model*. C++ projects generally contain at least two different types of file: header files (which tend to have a file extension such as `.h` or `.hpp`) and source files (which tend to have a a file extension such as `.cpp` or `.cxx`). Each source file can include multiple headers, which can themselves include other headers, and so on.

A C++ compiler compiles a program in pieces known as *translation units*. There is one translation unit for each source file in the program: a translation unit is essentially a source file that has had everything it includes inserted into it. Each translation unit is compiled separately and the result is stored in an *object file*. These object files are then passed to a separate program known as a *linker*, which combines them into an executable program that you can run. An illustration of this process is shown in Figure 7.1.

The question then arises as to how third-party libraries fit into this process. There are two main types of third-party library: *static* and *dynamic*. Static libraries (`.lib` files on Windows and `.a` files on Linux) can basically be thought of as something like a zip file that contains multiple object files. Using a static library is effectively equivalent to linking against the object files it contains. Dynamic libraries are slightly more involved. On Windows, these are split into two separate files: a `.lib` file, known as an *import library*, and a `.dll` file, known as a *dynamic link library*. Applications link against the import library, and then ensure that the dynamic link library can be found on the path at runtime. On Linux, dynamic libraries have the extension `.so` (standing for *shared object*) and applications directly link against them. Mac OS X is similar to Linux, except that dynamic libraries have the extension `.dylib`. For the purposes of this course, we will focus on Windows, but the process of using third-party libraries is reasonably similar on the other platforms.

7.3 Downloading and Building AngelScript

As mentioned, we will illustrate how to use a third-party library in Windows by using AngelScript as an example. The first thing to do is to download the AngelScript `.zip` file

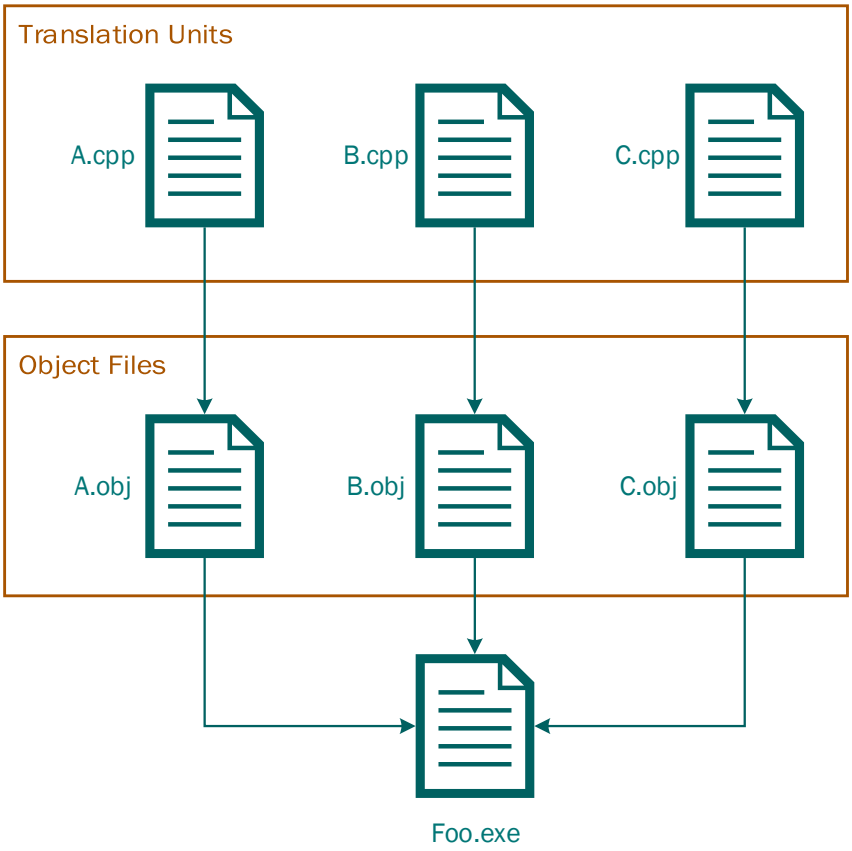


Figure 7.1: A simplified version of the C++ compilation model: individual translation units are compiled separately into object files, which are then linked together to form an executable program.

from the web and extract it into a directory.

Exercise 7.1: In your web browser, navigate to `http://www.angelcode.com/angelscript/downloads.html`. To download AngelScript, click on the large AngelScript SDK 2.31.0 link at the top of the page and save the resulting file (`angelscript_2.31.0.zip`) into your `Downloads` directory. Extract the contents of this file (a directory tree with a root directory called `sdk`) into a new directory called `angelscript_2.31.0` in the `Downloads` directory.

The AngelScript library is provided in source code form, so before we can use it we need to build it. This particular library is built using project files that have been provided for various different compilers. Unfortunately, other libraries will generally have different build procedures: you will need to read the documentation for each new library you want to use. In this case, we simply need to open up the solution file for Visual Studio 2013 and build the solution. That will create static libraries that we can link into our programs.

Exercise 7.2: Navigate to `angelscript_2.31.0/sdk/angelscript/projects/msvc2013` directory in Windows Explorer. This should contain a solution file called `angelscript.sln`. Open this, and build the solution. You should find the resulting static library in the `angelscript_2.31.0/sdk/angelscript/lib` directory: verify that this is the case.

7.4 Using AngelScript in Visual C++

Having built AngelScript, we can now use it in a project. We will first describe how to do this directly in Visual C++, before showing how to do it in CMake in the next section.

Exercise 7.3: As a first step, create a new Visual C++ project called `astest` with an empty `main.cpp` (refer back to §2.2 if you can't remember how).

To allow our program to use AngelScript, we need to specify two things in the property pages for our project: the location of the AngelScript header files, and the location of the AngelScript library. We also need to set our runtime library to match that used by AngelScript: this is a minor detail that is only needed for this particular library, and is not normally necessary.

Exercise 7.4: To access the property pages, right click on the `astest` project (not the solution) and click Properties. To specify the location of the header files, navigate to Configuration Properties → C/C++ → General and edit the field called Additional Include Directories. You need to add the full path to the `angelscript_2.31.0/sdk/angelscript/include` directory to this field. Now navigate to Configuration Properties → Linker → General, and edit the field called Additional Library Directories. To this, add the full path to the `angelscript_2.31.0/sdk/angelscript/lib` directory. Lastly, navigate to Configuration Properties → Linker → Input, and edit the field called Additional Dependencies. To this, add `angelscriptd.lib`.

As mentioned above, there is one final step we need to perform in the case of AngelScript: setting the runtime library to match that used by the library. To do this, navigate to Configuration Properties → C/C++ → Code Generation, and change the Runtime Library field to Multi-threaded Debug `/MTd`.

You should now be able to compile and run a simple program that uses AngelScript. To test this, add the following program to `main.cpp` and try to build it.

```
1  #include <iostream>
2  #include <angelscript.h>
3
4  int main()
5  {
6    asIScriptEngine *engine = asCreateScriptEngine();
7    if(engine) std::cout << "Successfully initialised AngelScript!\n";
8    engine->ShutDownAndRelease();
9    return 0;
10 }
```

If everything worked, you should see the message 'Successfully initialised AngelScript!'.

7.5 Using AngelScript in CMake

To use AngelScript in CMake, first set up a new CMake project called `ascmaketest` by following the instructions in §4.3, and generate a Visual C++ solution from it. One slight change is that this time you should use the Visual Studio 12 2013 generator (not the Visual Studio 12 2013 Win64 generator you used in §4.3): this is because the AngelScript libraries build as 32-bit by default. (An alternative would be to build AngelScript as 64-bit and use the Win64 generator, but we won't do that here.) If you try to build the application project in the generated solution, you will find that you get the following compiler error:

> Cannot open include file: 'angelscript.h': No such file or directory

This error is produced because you haven't specified the location of the AngelScript header files in the CMake script. To do this, add the following to your `CMakeLists.txt` file just before you specify the target, replacing the include path with the path to the AngelScript include directory on your own computer:

```
##########################################
# Specify additional include directories #
##########################################

INCLUDE_DIRECTORIES("C:/libraries/angelscript_2.31.0/sdk/angelscript/include")
```

If you now try to build the project again, you will find that it will now compile but not link. In particular, you will get the following linker error:

> unresolved external symbol asCreateScriptEngine referenced in function main

This error is produced because you haven't specified the location of the AngelScript library in the CMake script. To do this, add the following to your `CMakeLists.txt` file just after you specify the target, replacing the library path with the path on your own computer:

```
#################################
# Specify the libraries to link #
#################################

FIND_LIBRARY(AngelScript_LIBRARY angelscriptd HINTS
  ↪  "C:/libraries/angelscript_2.31.0/sdk/angelscript/lib")
TARGET_LINK_LIBRARIES(${targetname} ${AngelScript_LIBRARY})
```

If you build the project now, you will find that you get a different linker error, this time referring to a mismatched runtime library. To fix this, add the following to `CMakeLists.txt` above the place where you set the target name:

```
#####################################
# Specify additional compiler flags #
#####################################

SET(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} /MTd")
```

Having done this, the project should now build.

7.6 Further Reading

This chapter has introduced the basics of using third-party libraries in your programs. However, different libraries tend to be built in different ways, and many of them have their own dependencies that must be built before they can be built. As a result, it may be harder to use some libraries in your programs than others. Unfortunately, there is no easy answer to this: the best advice we can give is to try to use libraries that are easy to build, and, failing this, to try to find pre-built binary versions of the libraries that you can just use out of the box. Building software is something you get better at with practice, but it can still be arbitrarily difficult if the build for a particular library is badly set up. The web is a good source of advice for how to work around build issues: if you search for the error message you get, you will often find that someone has come across your specific problem before.

Separately, some of you may be interested more specifically in the AngelScript library we have seen in this chapter. This is well worth playing around with, and contains a lot of functionality. There is decent documentation available in the library manual: http://www.angelcode.com/angelscript/documentation.html. Separately, you might be interested in this article (written by one of the course authors) to simplify the process of calling functions in scripts by applying a bit of template hackery: http://accu.org/var/uploads/journals/overload95.pdf.

Chapter 8

Graphics Programming using SDL

8.1 Introduction

Thus far, you have been restricted to writing programs whose output is printed to the terminal in text form: needless to say, this is quite restrictive. In this chapter, we will start looking at how to produce more graphical output, using a third-party library known as SDL (this stands for Simple DirectMedia Layer). This will provide the background information you need to understand the mini-project you will be working on in Chapter 9.

8.2 Getting Started

To get you started, we have set up a simple solution containing an application that links to the SDL library but currently only contains an empty main function, a utility class called `ExecutableFinder`, and a library that provides some simple classes to handle mouse and keyboard input. The first steps are to clone the git repository and build the code.

Exercise 8.1: Clone and build the code using the following commands:

```
$ git clone https://bitbucket.org/gpcwm/sdlsandbox.git
$ cd sdlsandbox
$ ./build-win.sh 12 Debug
```

Having done this, open up the Visual Studio solution file (`sdlsandbox.sln`) in `sdlsandbox/build`. You will note that it contains two main projects: an application called `sdlsandbox` and a library called `tvginput`. The project you will be modifying is `sdlsandbox`; the `tvginput` library is only there to make input-handling easier.

8.3 Initialisation and Shutdown of SDL

Before we can use SDL, it must be initialised using a function called `SDL_Init`:

```
1 if(SDL_Init(SDL_INIT_VIDEO) < 0)
2 {
3         // Signal an error
4 }
```

This should be done before calling any other SDL functions (a good place for it is usually somewhere near the start of the `main` function). For reasons of cleanliness, SDL should also be shut down again in the following way before your program terminates:

```
1 SDL_Quit();
```

Exercise 8.2: Add the indicated code to initialise and shut down SDL to your `main` function. What should your program do if SDL fails to initialise? Add appropriate code to handle this case.

8.4 Setting up the Event Loop

The next step is to set up the application's event loop. This repeatedly polls SDL for any events that have occurred (e.g. key presses by the user), and handles them appropriately. To keep the code clean, we're going to create an `Application` class to encapsulate the event loop and associated input handling.

Exercise 8.3: As a first step, add two new source files, `Application.cpp` and `Application.h`, to the project to contain the implementation of the new class.

As a reminder, to do this in a CMake project, you need to first create the files on disk, and then open up the CMakeLists.txt file for the project, specify the files and regenerate the project using CMake (you can also just try to build the project, in which case the CMake plugin will handle the regeneration process).

Having added the source files, we now need to implement the `Application` class. To get started, add the following class interface to `Application.h`:

```
1  #ifndef H_SDLSANDBOX_APPLICATION
2  #define H_SDLSANDBOX_APPLICATION
3
4  // Prevent SDL from trying to define M_PI.
5  #define HAVE_M_PI
6
7  #include <SDL.h>
8
9  #include <tvginput/InputState.h>
10
11 class Application
12 {
13 private:
14   /** The current state of the keyboard and mouse. */
15   tvginput::InputState m_inputState;
16
17 public:
18   /** \brief Runs the application. */
19   void run();
20
21 private:
22   /** \brief Handles key down events. */
23   void handle_key_down(const SDL_Keysym& keysym);
24
25   /** \brief Handle key up events.*/
26   void handle_key_up(const SDL_Keysym& keysym);
27
28   /** \brief  Processes SDL events (e.g. those generated by user input).
29       \return Whether the application should continue. */
30   bool process_events();
31 };
32
33 #endif
```

To implement the event loop, you need to add code to the `run` and `process_events` functions. The `run` function should repeatedly call `process_events` until it returns `false`. The `process_events` function should loop over and process any events that have occurred, using the `SDL_PollEvent` function in SDL to poll for events. A simple way to process an individual event is to switch on the `type` field of the event.

Exercise 8.4: Implement a basic first version of `run` and `process_events`. Initially, you should ignore most of the events and simply worry about handling events whose type is `SDL_QUIT`. If one of these is received, `process_events` should return false; in all other cases it should return true.

The next step is to handle keyboard events. To do this, you will need to implement the `handle_key_down` and `handle_key_up` functions, and then make sure they are appropriately called in `process_events`.

Exercise 8.5: Implement the two key-handling functions and hook them into the overall event loop. They should be passed the `SDL_Keysym` object contained in a keyboard event: this can be accessed using `event.key.keysym`. The functions should appropriately update the application's input state (stored in the `m_inputState` member variable). To get the `Keycode` you need to pass to `InputState`'s member functions, you can simply cast `keysym.sym` to `Keycode`. To test that you've done this correctly, you should now do two things. First, add the following code to `main` to run the application:

```
1 Application app;
2 app.run();
```

Second, modify `Application::run` so that the application exits when you press the escape key (`KEYCODE_ESCAPE`). If you now run your application, you should be able to press escape to exit: verify that this is indeed the case.

Question: What advantages does SDL's event-based approach to input handling have over directly querying hardware devices such as the keyboard?

8.5 Creating a Window

Having got a basic event loop set up, the next step is to create a window in which to draw things. To keep the code tidy, we're again going to add a new class, this time called `WindowedRenderer`, which will handle the management of the window and any rendering we do into it.

Exercise 8.6: Add two new source files to the project, `WindowedRenderer.cpp` and `WindowedRenderer.h`, to contain the implementation of the new class.

Having added the source files, we now need to implement the `WindowedRenderer` class. To get started, add the following class interface to `WindowedRenderer.h`:

```
1 #ifndef H_SDLSANDBOX_WINDOWEDRENDERER
2 #define H_SDLSANDBOX_WINDOWEDRENDERER
3
4 #define HAVE_M_PI // prevent SDL from trying to define M_PI
5 #include <SDL.h>
6
7 #include <boost/shared_ptr.hpp>
```

```
8
9  class WindowedRenderer
10 {
11 private:
12   typedef boost::shared_ptr<SDL_Surface> SDL_Surface_Ptr;
13   typedef boost::shared_ptr<SDL_Window> SDL_Window_Ptr;
14
15   /** The height and width of the window. */
16   int m_height, m_width;
17
18   /** The window into which to render. */
19   SDL_Window_Ptr m_window;
20
21   /** The SDL surface associated with the window. */
22   SDL_Surface *m_windowSurface;
23
24 public:
25   /** \brief Constructs a windowed renderer.
26       \param title The title of the window. */
27   explicit WindowedRenderer(const std::string& title);
28
29   /** \brief Renders the contents of the window. */
30   void render() const;
31 };
32
33 #endif
```

To implement the windowed renderer, you need to implement the constructor and the `render` function. For now, you should just implement a blank `render` function and focus on the constructor. The constructor should both create the window and get its associated `SDL_Surface` (this will be needed to actually draw anything to the window). The window can be created by making a call to a function called `SDL_CreateWindow`. To ensure that the window is properly cleaned up later in the program, we need to make sure that SDL's window destruction function, `SDL_DestroyWindow`, is called on the window when the `WindowedRenderer` containing it is destroyed. A conventional way of doing this would be to add a destructor to `WindowedRenderer` that explicitly calls `SDL_DestroyWindow`, but shared pointers give us a cleaner way of achieving the same goal: specifying a deletion function that should be called on the managed pointer when the shared pointer's reference count goes to zero. In this case, we store our window as a shared `SDL_Window` pointer, `m_window`, within `WindowedRenderer`. When `WindowedRenderer` is destroyed, the reference count for the `m_window` shared pointer it contains will decrease to zero, which will cause the deletion function associated with the shared pointer to be invoked. We can thus arrange for our window to be appropriately destroyed later as follows:

```
1 m_window.reset(SDL_CreateWindow(/* params here */), &SDL_DestroyWindow);
```

One thing to note when passing the title of the window to `SDL_CreateWindow` is that the `title` parameter to the `WindowedRenderer` constructor is a `std::string`, whilst `SDL_CreateWindow` expects a `const char *`. To get a `const char *` from a `std::string`, you can use the `c_str` member function of `std::string`.

Once the window has been created, `SDL_GetWindowSurface` can be used to get a pointer to its associated surface for storage in `m_windowSurface`.

Exercise 8.7: Implement the constructor for `WindowedRenderer` as described above. To actual create the window, you need to add a `WindowedRenderer` to the `Application` class. To do this, edit `Application.h` to include `WindowedRenderer.h`, and add the following private member variable to the `Application` class:

```
/** The renderer. */
boost::shared_ptr<WindowedRenderer> m_renderer;
```

Also add a public constructor with no parameters:

```
Application();
```

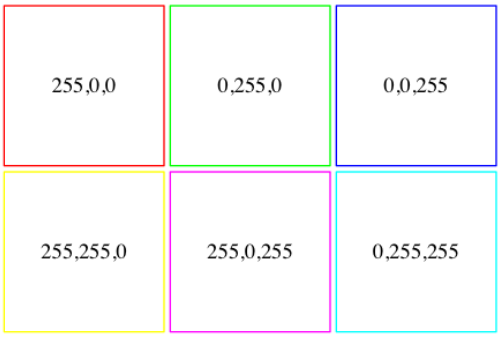
Then, in `Application.cpp`, add an implementation of this constructor that creates the `WindowedRenderer`:

```
Application::Application()
{
  m_renderer.reset(new WindowedRenderer("SDL Sandbox"));
}
```

Finally, make sure that the renderer's `run` member function is called from `Application::run`. Try running your program, and make sure that the window is created and rendered as you expect. If you're interested, you can experiment with some of the parameters to `SDL_CreateWindow` to see what they do.

8.6 Basic Rendering

The window we have just created contains a drawing area that conceptually consists of a rectangular grid of square picture elements (*pixels*). The colour of each individual pixel is determined by the contents of a buffer associated with the window's surface. The pixels in the buffer are stored in memory row-by-row, top-to-bottom and then left-to-right (this is known as *raster order*). Assuming a standard 24-bit RGB format for the window, each pixel will have an 8-bit (0-255) value for each of its colour components (red, green and blue):



Drawing things into the window then ultimately boils down to setting the colour components of individual pixels.

To display the window, the monitor needs to render the contents of the buffer onto the screen many times per second. This poses a problem when the contents of the window are changing: since it is not possible to update all of the pixels in the buffer simultaneously, the window rendered by the monitor will spend most of its time showing part of one frame and part of the next, which can cause undesirable effects such as tearing. A separate problem is that it is often much easier to simply clear the window before rendering new objects than it is to somehow try and track old objects in order to overwrite them: however, if we clear the window whilst the monitor is in the process of drawing it and then start rendering new objects again, it can lead to flickering.

Both of these problems can be solved by a technique called *double-buffering*: the idea is to write the desired contents of the new frame into a separate buffer, known as the *back buffer*, wait for the monitor to finish rendering the window and then quickly copy everything

across from the *back buffer* to the normal buffer, now known as the *front buffer*, before the monitor starts the next pass. Rendering in SDL implements this strategy: all the rendering calls you make will write into the back buffer, and so at the end of each rendering pass you must call a special function, `SDL_UpdateWindowSurface`, to copy everything across to the front buffer for display on the screen.

To test out some basic rendering, we will experiment with drawing coloured rectangles on the screen. More interesting rendering can be performed using SDL's 2D accelerated rendering API, which has functions for drawing things like lines. In the simple API, the function to draw filled rectangles is called `SDL_FillRect`.

Exercise 8.8: Add code to `WindowedRenderer::render` to draw a filled red rectangle and then perform the buffer copy. By making appropriate changes to `Application` and `WindowedRenderer`, can you find a way to allow the user to control the position of the rectangle using the arrow keys on the keyboard?

8.7 Drawing Sprites

Drawing simple primitives is all well and good, but it's often useful to be able to draw more structured entities such as 2D sprites. In this section, we will look at how to load such sprites from bitmap files on disk, and how to draw scaled versions of them in the window (using a process known as *blitting*). If you look in the `sdlsandbox/apps/sdlsandbox/resources` directory, you will find a bitmap file called `apple.bmp`, which contains the sprite we are going to load and render in this exercise.

SDL provides the ability to load bitmap files as part of its core functionality: this is accessed via the `SDL_LoadBMP` function, which takes the name of the bitmap file to load and returns a pointer to an `SDL_Surface` containing the image. To render a scaled version of a loaded sprite from such a surface to the window, we simply need to call `SDL_BlitScaled` and specify the destination rectangle in the window. However, for this to work, the formats of the sprite surface and the window must be the same: to ensure this, we can convert the sprite surface's format to that of the window using a function called `SDL_ConvertSurface`.

A remaining difficulty lies in specifying the location of the bitmap file on disk in a way that does not rely on the directory from which the program was launched. It is certainly possible to specify an absolute path to the file, but programs that do this are hard to move between different machines. A better approach is to specify a path relative to the location of the program's executable, which can be determined by the `ExecutableFinder` class that has been provided to you.

Exercise 8.9: First, add a `load_bitmap` function to `WindowedRenderer` that has the following signature:

```
1 SDL_Surface_Ptr load_bitmap(const boost::filesystem::path& path) const;
```

Note that you will need to include `boost/filesystem.hpp` in `WindowedRenderer.h` to get this to compile. Next, implement this function: the desired behaviour is for it to load the specified bitmap and convert the format of the loaded surface to that of the window. Finally, add a member variable of type `SDL_Surface_Ptr` to store the sprite, load it in the constructor (being careful to specify the path relative to that of the program's executable) and use `SDL_BlitScaled` to render it in the `render` function before the buffer copy. Run the program to verify that the sprite is being correctly drawn. Can you think of a way to make the sprite randomly move around the window, changing its dimensions as it goes?

8.8 Further Reading

Due to the time constraints of this course, this chapter has necessarily only scratched the surface of graphics programming using SDL. Interested students may want to look into some of the many SDL tutorials on the web, or acquire a copy of the book *SDL Game Development*, by Shaun Mitchell, which goes into substantially more depth than we are able to here.

Chapter 9

Mini-Project 2: Nibbles

9.1 Game Overview

Nibbles (also known as *Snake*) is a classic one-player game in which the player controls a snake as it tries to finds its way out of a maze, eating apples as it goes. The playing area for the game is represented using a grid of square cells. At each discrete time step, the snake moves by a single cell in the direction specified by the user (up, down, left or right). The snake is initially small, but grows in length by a single cell every time it eats an apple. If the snake collides with a wall in the maze, it dies, and the player loses the game. The exit to the maze opens when the snake reaches a certain length. The player wins by guiding the snake successfully to the exit. Figure 9.1 shows a screenshot from an example run-through of the game, in which a snake that currently consists of 4 cells (shown in red) is just about to eat an apple and become length 5.

In this mini-project, you will be given a partial implementation of *Nibbles* and asked first to complete it, and then to extend it to support an additional two-player deathmatch mode. If there is time, you will also be asked to look into the random generation of game worlds.

9.2 Getting Started

The first steps are to clone the git repository and build the code.

Exercise 9.1: Clone and build the code using the following commands:

```
$ git clone https://bitbucket.org/gpcwm/nibbles.git
$ cd nibbles
$ ./build-win.sh 12 Debug
```

Having done this, open up the Visual Studio solution file (`nibbles.sln`) in `nibbles/build` and study the code. This solution is much larger than the ones we have seen previously, and is divided into multiple projects. The code to simulate games of *Nibbles* is in a library called `nibbles`. On top of this are built two separate applications, `nibblescli` and `nibblesgui`, which both link to `nibbles` and make use of its functionality. The `nibblescli` application contains a command-line interface to `nibbles` that can be used for testing purposes. The `nibblesgui` application contains the main graphical interface to `nibbles`, as illustrated in Figure 9.1.

9.3 The Existing Code

The `nibbles` library is divided into a number of directories:

- The `games` directory contains classes representing the different types of *Nibbles* game that can be played. Your initial version of the code currently only contains the `Game` base class and a derived class called `Escape` that represents the one-player version of *Nibbles* described in the previous section.

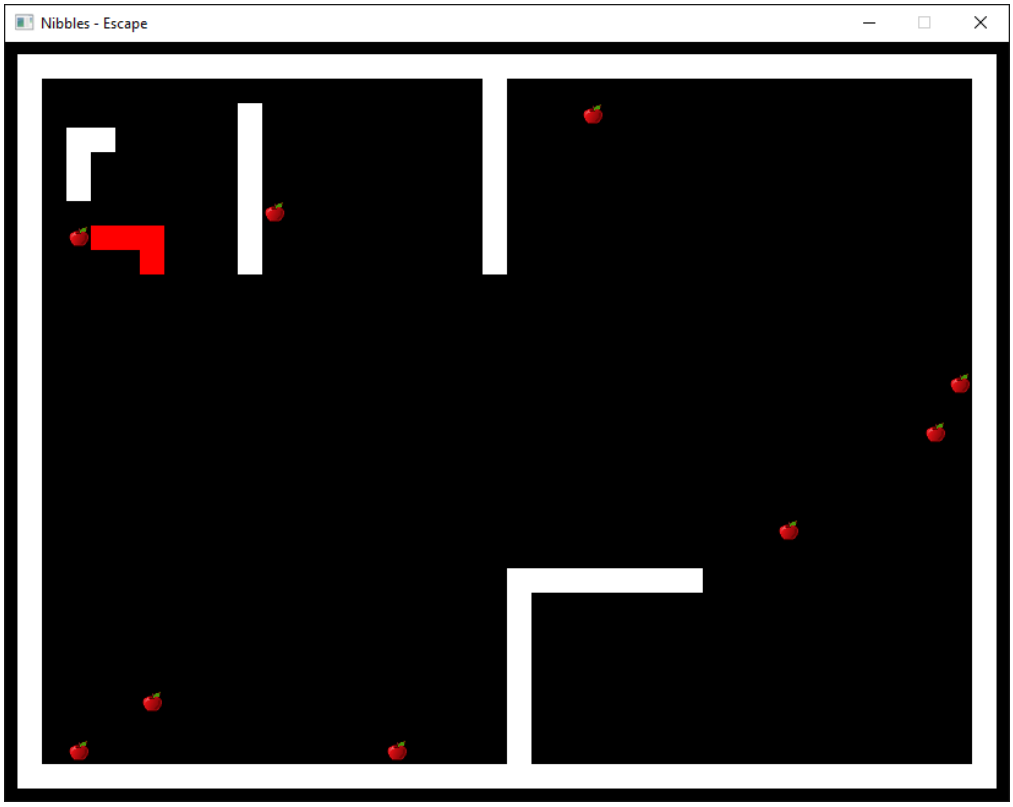


Figure 9.1: A screenshot from an example run-through of *Nibbles*: a snake that currently consists of 4 cells (shown in red) is just about to eat an apple and become length 5.

- The `model` directory contains the classes that model aspects of the *Nibbles* world such as the playing area, game objects and collision handling. Your version of the code contains a `PlayingArea` class to represent the game's grid-based playing area, classes such as `Snake` and `StaticObject` to represent objects in the world, and various collision handlers that can be used to control what happens when objects of different types collide. You will also see an `ObjectDatabase` class, which is used to keep track of all of the objects in the world, and a `World` class, which collects together the playing area, object database and collision handlers and simulates the *Nibbles* world.
- The `util` directory contains utility classes: `ExecutableFinder` can be used to find the location of the current program on disk, which is useful when trying to locate resource files; `IDAllocator` manages numeric IDs, and is used to number the different objects in the game.

Spend a bit of time familiarising yourself with the code to make sure you understand how it works before moving on to the next section.

9.4 Completing the Escape Game

Object Movement

If you run your version of the *Nibbles* code (`Ctrl + F5` by default in Visual Studio), you will see that it doesn't currently do very much: the initial state of the game will be rendered, but none of the objects will move. This is caused by the missing implementation of the `run_step` function in the `World` class, which is intended to simulate a single step of the *Nibbles* world. If you look at the existing contents of `run_step`, you will see that it currently contains comments outlining the desired behaviour of the function, but no actual code: your job is to implement the code to get the game working.

Exercise 9.2: First implement code to iterate over all of the objects and make the moves they choose for themselves. To determine the move an object wants to make, you can call its `choose_move` function; to actually move the object, you can call its `apply_move` function. Some of the moves chosen may lead to collisions between objects, which in some cases should cause the subsequent removal of some of the objects from the world, but at this stage you should not worry about either of those things: we will discuss them in the next section.

Collision Handling

Assuming you have correctly implemented the functionality described, the player's snake will now move when you run the game. However, because we have not as yet made any attempt to properly handle collisions between objects in the world, you will find that the snake simply keeps travelling through both walls and other objects: indeed, it is currently possible for it to escape the bounds of the world and cause the game to crash. To fix this, we must adapt `run_step` to detect collisions between objects and respond to them appropriately.

To understand how to detect collisions, we must first look at what can cause them, which involves looking at how moves are represented in the existing code. Currently, every move is an instance of the `Object::Move` struct, and consists of three things: an ID indicating the object to be moved, an optional world cell that the object may want to occupy, and a list of world cells that the object no longer wants to occupy. Collisions thus result from an object trying to occupy a cell that is already occupied by another object. To check whether a particular move causes a collision, we can thus simply look at the world's playing area to determine whether the cell that the object is trying to occupy (if any) is currently occupied by another object, and which object if so.

Responding to collisions is slightly more involved, because it depends on the types of the objects concerned. The way we implement this in the existing code is to register a specific handler for each potential type of pairwise collision that can occur: for example, if a snake collides with a wall then it should die, whereas if it collides with an apple then it should consume the apple and increase in length. We distinguish between the two objects involved in a collision: the moving object (the one that causes the collision) is denoted the *collider*, and the other is denoted the *collidee*. The collision handlers are stored in `World::m_collisionHandlers`, a map from object type pairs to collision handlers, and registered in the constructor of each game type (e.g. see `Escape::Escape`). There are three types of collision handler currently implemented in the code:

- The `ConsumeCollisionHandler` class implements a collision handler that kills the collidee and prevents any cells from being removed from the collider. For example, this kind of handler is used when a snake collides with an apple: the apple is consumed and removed from the world, and the snake grows in length by a cell as a result of adding the cell that contained the apple to its head, and not removing any cells from its tail.

- The `DieCollisionHandler` class implements a collision handler that kills the collider and prevents any move from taking place. For example, this kind of handler is used when a snake collides with a wall.

- The `KillCollisionHandler` class implements a collision handler that kills the collidee, but allows the move as long as the collider and the collidee are not the same object. For example, this kind of handler is used when a snake collides with another snake (or itself).

To respond to a collision, we need to look up the relevant handler and use it to handle the collision. As mentioned, resolving a collision can result in the death of one (or in principle both) of the colliding objects: these must be carefully removed from the world to prevent them from interfering with the rest of the game.

Exercise 9.3: Extend your implementation of `World::run_step` from the previous section to handle collisions and appropriately remove dead objects from the game world. Note that it is possible to kill an object that has not yet been simulated in the current step: be careful in your implementation not to simulate such objects when they are already dead.

9.5 Two-Player Deathmatch Mode

Your current version of the code currently only contains a single game class, called `Escape`, which represents a one-player version of *Nibbles* in which the player tries to grow their snake to a length sufficient for the exit to open so that they can escape. However, other types of *Nibbles* game are also possible. In this section, we will look at implementing a two-player version of *Nibbles* in which two players control snakes that try to bite one another: assuming neither player collides with a wall or themselves first, the player whose snake bites the other first wins the game.

The intended game mechanics are relatively straightforward. Instead of having one snake, you have two, controlled by two players pressing different sets of keys on the keyboard (we recommend the arrow keys for Player 1, and WASD for Player 2). Unlike in the `Escape` game, there is no exit: the two snakes must battle it out until one of them wins.

Exercise 9.4: As a first step, copy the `Escape` class to make a new class called `Deathmatch` in the `games` directory. Adapt it to support two snakes, remove any functionality related to the exit, and change the win conditions specified in `Deathmatch::run_step_post` appropriately.

Since one of the key ways to lose in a deathmatch game (aside from colliding with a wall or yourself) is to be bitten by the other snake, there is an advantage to being small (and thereby presenting less of a target). To support this concept, we will introduce a new type of object: *poisoned apples*. Poisoned apples are exactly like normal apples, except in their effect on a snake that collides with them: instead of increasing the snake's length by a cell, they decrease it by a cell (in implementation terms, we will need a new collision handler that removes an extra cell from the snake's tail). If removing an extra cell causes the snake's length to decrease to zero, the snake dies. Players must therefore trade off the need to remain as small as possible with the need to avoid eating poisoned apples when they are too small. The code you have already loads the relevant graphic for a poisoned apple and renders objects whose typed is 'PoisonedApple' correctly; however, you will need to implement a new collision handler yourself.

Exercise 9.5: Add a new collision handler called `ConsumePoisionCollisionHandler` to the `model/collisions` directory that implements the functionality just described. The collision handler will need to be registered in the constructor of `Deathmatch`. To test the poisoned apple functionality, adapt the code in `Deathmatch::run_step_post` to randomly add either apples or poisoned apples to the world.

9.6 Challenge: Random World Generation

Currently, the worlds for *Nibbles* games are loaded from text files in the `nibblesgui/resources` directory. This is effective, but making such worlds can be quite labour-intensive: as a result, people are unlikely to make many worlds for the game, leading to a lack of variety in gameplay.

To ameliorate this problem, it would be nice to add a way to generate *Nibbles* worlds randomly, but this is not entirely straightforward. The 'obvious' approach might be to first randomly generate the walls, and then add objects at random points in empty space, but this does not work very well in practice: 'good' *Nibbles* worlds should ideally contain wall structures that look plausible but do not 'wall off' parts of the world, and randomly-generated walls tend to satisfy neither of these constraints.

Your challenge in this section is to design and implement a sensible random world generator that satisfies the design constraints just mentioned. We are deliberately not going to provide any written guidance as to how you should go about this, because it would ruin the fun (we want to see what you can come up with!). However, your demonstrators will be happy to discuss your ideas with you and to try to help you refine them.

Exercise 9.6: Implement a random world generator for *Nibbles*. Chocolate is available for the best solution.

Chapter 10

Closing Remarks

We have covered quite a lot of ground in this course, and hopefully had a certain amount of fun. However, the course has a serious purpose as well: to provide you with good C++ and software development skills that you can apply throughout the rest of your degree. To conclude the course, therefore, we want to provide you with a few things to think about when you're writing code in the future. Some of these reiterate things we've talked about already during the course; others are new.

Readability Programming languages are a communication medium. When you write programs, you're communicating with two audiences: the computer and other programmers (including potentially yourself in six months' time). The computer doesn't really care how you write code (as long as it's syntactically-correct), but other programmers do. As such, you should aim to write clear, readable code that accurately communicates your intent:

- Use clear names for variables, functions, classes, etc.
- Comment your code appropriately.
- Limit the complexity of individual functions and classes: if a function or a class is getting too big, split it into multiple pieces.

Software Development Logistics If your software is hard to set up on a new machine and hard to build, no-one will want to work on it with you. The very first thing you should do when you start a major new project is to set up a automated single-step build on all the platforms in which you're interested, and test it on lots of different machines to make sure it actually works. This will probably take a day or two, but it will save you time later.

Software Architecture Software systems can be incredibly complex, and this complexity increases non-linearly as the size of a system increases. As such, it is important to structure your code in such a way as to make it as easy as possible for your brain to understand. Here are a few tips:

- One of the things that can make your code hard to understand is the sheer amount of code you have. The best solution to this problem is to delete some of it. Often, you will find you have several bits of code that do the same (or almost the same) thing: if you factor out the commonality between them, you can often reduce the overall amount of code you have. Just as importantly, avoid making the problem worse by duplicating code in the first place: instead of copying and pasting code from one place to another, think about how to factor out the commonality into something more reusable.
- Try to isolate code that uses a class from the way in which that class is implemented internally. The smaller the number of things that directly depend on a piece of code, the easier that piece of code is to change.
- Structuring your code hierarchically can make it easier to understand: don't just group functions into classes, group classes into packages (directories in C++) and packages

into libraries. Hierarchical projects are easier for people to understand, because they can think about the high-level structure of your code without having to worry about all the tedious low-level details unless they're really interested.

- In almost all cases, your code will improve if you avoid/remove cyclic dependencies (both at the class level and more broadly). Cyclic dependencies make individual components of your code harder to reuse and harder to test.

For more information on software architecture, we recommend *Large-Scale* `C++` *Software Design* by John Lakos. It's quite an old book now, and some of its advice is out-of-date, but it still contains a lot of very good information, particularly on cyclic dependencies.

Optimisation There are a number of keys to making your code run fast:

- Profile your code to find out what's taking all the time! Humans are dismal at guessing why their code is slow; profilers are really good at telling them.
- Use appropriate algorithms, taking into account their computational complexity and the size of the problem at hand. You can generally achieve much bigger speed-ups by switching to a better algorithm than you can by fine-tuning the implementation of the algorithm you currently have.
- Rethink your data structures, taking into account things such as the way your cache works. For example, traversing a `std::list` can be a lot slower than traversing a `std::vector`, because the elements of a list are not contiguous in memory.
- Think about parallelising your code. If you're running everything on a single core when you have multi-core machine, there's a huge amount of computational power potentially going to waste.

Importantly, it's much easier to optimise readable code than it is to make optimised code readable. Don't needlessly compromise readability: wait until your profiler tells you that you need to optimise something, and then optimise it.

Teamwork Whilst it's possible to write software on your own, there will generally be times when you're working in a team. This can lead to all kinds of challenges that are beyond the scope of this course. That said, here are a few thoughts:

- Try to make your programs as simple as possible. 'Cuteness' is not a desirable property of code. As Dijkstra put it:

  > The competent programmer is fully aware of the limited size of his own skull. He therefore approaches his task with full humility, and avoids clever tricks like the plague.

- Most good projects on which you work will follow some sort of coding style. Stick to it – it may not be optimal (it's not even clear how you evaluate optimality), but the conceptual integrity of the code is important.
- Review code, not people. Conversely, your code is not you.
- Team projects have a 'truck number' – this is the number of people who would need to be hit by a truck in order to derail the project. A truck number of 1 is not a good thing: try to share knowledge between the different people working on your project.
- People in a team have different skills: try to figure out what everyone's good at, and make sure they have the opportunity to do that.
- Always try to work for the team, not yourself. A team of weaker individuals with aligned goals will always beat a team of stronger individuals who all want different things.

That's all folks! Best of luck in your coding endeavours, and never forget that programmers run on cookies.