Low Fat C++ for Engineers

Konstantinos Lappas

The C++ Programming Language

C++ is a general-purpose programming language originally created by Bjarne Stroustroup. The initial concept was to create an extension to the C programming language that would assist object-oriented programming through the use of *classes*. The programming world soon realized the potential of the new language and so C++ took off on its own, gaining its place in the software industry and eventually its own standardization division in the *International Standardization Organization (ISO)*.

C was developed in the early 70s and it was designed to be light and simple and close to the metal as possible. This made it extremely popular because although it was a general purpose and humanly readable, it created fast and efficient code. C++ started up around 1980 and was publicly available in the mid-80s. It was built to be compatible with C and take advantage of its best characteristics, while offering some extra features that would make software development easier and more robust.

Over the years the compilers for these languages have evolved dramatically producing code of exceptional quality. Apart from that the languages have evolved by a great margin as well. The standardization committees have proposed changes that give them some fantastic characteristics that make the code compact and mistakes harder to make.

C++ is one of the 'Higher-Level languages' as we call them. In the early days of computers people were writing code in *assembly language*, which consists of direct commands to the processor. That kind of programming requires a lot of repetitive work and above all the program cannot be moved to another system. The advent of 'Higher-Level languages' has made things a lot easier, since these languages are modeled after human understanding and are more accessible. Another benefit is the portability of code because all you must do is create the translator for each language for every system. Then code portability is a straightforward task. These translators are the compilers and the interpreters.

The compilers read the source code and convert it to machine code all at once. Interpreters on the other hand read the source code one line at a time and execute it. We cannot say that one language is compiled or interpreted since there are both versions for most of them. Traditionally though some languages are compiled, and others are interpreted.

C and C++ are compiled while Python is interpreted.

For the purposes of this book we will work on Windows 10, using Visual Studio 2019 Community Edition.

C++ is a quite simple language. You can learn the basics and create a valid program in a noticeably short time. Learning the most complicated features of the language surely takes

a while because you need to accumulate programming experience to completely understand and appreciate them.

One of the things C++ inherited from C is the lack of any built-in functionality. There is no 'command' to print anything on the screen or read any input from the keyboard. All these tasks are imported into our programs from external libraries. The standardizations committee has made sure that trivial operations like reading the keyboard or printing to the screen or file input and output, as well as a great number of useful functions is in every implementation of the language and lies within the standard library. The structure of this library is strictly defined, and you can be sure that is cross platform.

The code we develop in C++ is stored in text files usually referred to as 'source' files. The compiler reads these files and translates their contents into machine code instructions. Then we use another program called 'linker'. The linker takes the output files generated by the compiler, check for libraries that might be needed and puts all the machine code in one file which is the executable file. This file contains everything required by the system to do what we intended.

In everyday life the first time we compile and run our programs they do anything but what we intended. This is natural bearing in mind that it was a human that built it. Unless your program does nothing but display a greeting to the user, but does something more complex instead, chances are that something will go wrong. As programs grow bigger and more complex and more people are involved in their development the number of errors or bugs as we usually call them, grows.

These bugs trigger the so-called debugging cycle. Every time we complete the development of a feature in our software, and we build the executable, we perform a series of tests to make sure that everything runs as expected. Not only do we test the new features but the old ones as well, to make sure we have not broken anything. At the end we collect the errors we have found; we design and implement solutions and we start testing again. This cycle is repeated until we are satisfied with the result.

Creating bug free code is not an easy task. There are many methodologies proposed over the years. The iterative process described before is the simplest one to use. It is very straight forward and if we design a complete set of tests and routinely perform them we can have a particularly good result. There are several well-established methodologies for developing robust software, but I should have to author another book just to scratch their surface. So, I will stick to this simple method which is adequate for the code in this book and more than enough to get you started with the idea of creating stable software. Going deeper into this topic is far beyond the scope of this book.

Chapter 1: Basic concepts

The objective of this unit is to see the base elements of the language.

Before we write our first program it would be nice to see some fundamental features of the language. That will help us more easily understand the structure of code. These features are the variables and the functions.

Variables

Variables are used to store data. Their key feature is that the values of the data stored can be modified hence the name 'variable.' Creating a variable in C++ is simple. All you must do is specify its type and its name. The compiler needs to know how much space to allocate for the variable since different types require different amount of space.

Here are the fundamental variable types we have in C++

- *int* : integer numbers, usually takes 4 bytes, and can store values from -2147483648 to 2147483647
- *float* and *double* : floating point numbers, float takes 4 bytes and double 8. Double has two times the precision of float
- *char* : 1-byte long character char
- *bool* : 1-byte long Boolean variable, can be either *true* or *false*.

Variable types can be modified using *C++ type modifiers* These are:

- signed
- unsigned
- short
- long

They apply to the following types:

- int
- double
- char

Here is the list of the modified data types

data type	size in bytes	description
signed int	4	integers, equivalent to int
unsigned int	4	only positive integers
short	2	small integers, range -32768 to 32767

long	at least 4	large integers, equivalent to long int
unsigned long	4	only positive long integers
long long	8	really large integers
unsigned long long	8	really large positive integers
long double	at least 8	large floating-point numbers
signed char	1	characters, range -128 to 127
unsigned char	1	characters, range 0 to 255

Here are some examples of variables

```
int a = -135;
unsigned int b = 782;
long c = 2837465;
double d = 1.234;
char e = 'a';
```

In C++ we also have data types that are derived from these, as well as user defined types. *Arrays*, *pointers*, and *structures* are some of these types. We will cover them later in this section.

Constants

C++ introduced the concept of *constant.* This is similar to the variables we saw before, only their value cannot be modified. It is set upon initialization and remains constant throughout its lifespan.

Constants can be used anywhere within the code a constant value is expected.

```
const int c = 10;     // const variable
const int& cr = c;    // reference to const variable
const int* cp = &c;   // pointer to const variable
// the following statements generate errors
// because we  try to modify constant data
c = 1;
cr = 2;
*cp = 3;
```

Constants are assigned their values when they are declared at their initialization, because any other time it is considered assignment.

const pointer

After the pointer to constant variable, we should see the constant pointer.

```
data_type* const pointer_name = &variable;
```

We can modify the contents of the pointer, the value of the variable in other words, but we cannot make the pointer point to another variable.

```
int i,j;
int* const pi = &i;  // constant pointer initialized
*pi = 10;            // ok to change content
pi = &j;             // error, constant pointer
```

Operators

Data manipulation is core functionality for every programming language. We need to combine and store data for later use. These manipulations give value to the information we have access to.

The languages provide us with *operators* that help us manipulate our data. The basic operator is the *assignment operator*. This is used to assign a value to a variable, either directly from a value or copying data from another variable.

The operators are symbols that represent mathematical or logical operations. C++ has a rich repertoire of operators letting us manage our data with ease. Operators can be divided into categories depending on their nature.

- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise operators
- Assignment operators
- Misc (these include *sizeof* operator, *comma* operator and *conditional* operator among other)

Operators are evaluated in strict order defined by the standard. This order is called operator precedence. For arithmetic operators' precedence is the same as in mathematics.

First, we will go through the operators available in C++ and then we will examine their order of execution.

Arithmetic operators

Operator	Description	Syntax
+	Add two operands	a + b
-	Subtract two operands	a - b

*	Multiply two operands	a * b
/	Divide two operands	a / b
%	Modulus operator, returns the remainder of the integer division	a % b
++	Prefix / postfix increment	++a / a++
--	Prefix / postfix decrement	--a / a--
-	Unary minus, changes the sign of the numeric value	-a

Relational operators

Operator	Description	Syntax
==	Checks if two operands are equal or not, if they are returns true, otherwise false	a == b
!=	Checks if two operands are equal or not, if they are NOT returning true, otherwise false	a != b
>	Checks if the value on the left is greater than the value on the right	a > b
<	Checks if the value on the left is less than the value on the right	a < b
>=	Checks if the value on the left is greater than or equal to the value on the right	a >= b
<=	Checks if the value on the left is less than or equal to the value on the right	a <= b

Logical operators

Operator	Description	Syntax
&&	AND operator, returns TRUE if both operands are TRUE	a && b
||	OR operator, return TRUE if either of the operands is TRUE	a || b
!	NOT operator, negates logical status of expression	!a

Bitwise operators

Operator	Description	Syntax
&	AND between each bit	a & b
|	OR between each bit	a | b
<<	Shift bits to the left as many places as the second operand indicates	a << b
>>	Shift bits to the right as places as the second operand indicates	a >> b
~	Invert all the bits of the value following	~a
^	XOR between each bit	a ^ b

Bitwise operators perform on bit level. This makes them hard to understand unless you understand binary arithmetic. So here is a brief description.

First, we assume we have two numbers, a=5 and b=11. In binary they are a=00000101 and b=00001011. We are using binary representations because of the clarity they offer in this case.

- AND : a & b = 00000001, only the last bit is 1 in both operands, in decimal this is 1.
- OR : a | b = 00001111, at least one operand has 1 in one of the last 4 digits, decimal 15.
- Left shift : a << 2 = 00010100, all bits are shifted by 2 positions, same as multiplying by 4 which is 2^2. The base 2 comes from binary and the exponent is the shift operation second operand. In decimal the result is 20.
- Right shift : a >> 2 = 00000001, all bits are shifted right by two positions. This is the same integer division by 4. The decimal result is 1.
- Invert bits : ~a = 11111010, all bits are inverted. The decimal result is 250.
- XOR : a ^ b = 00001110, exclusive OR is true when one operand NA ONLY one is true, in our case bit. The first four bits of our numbers are 0 and the last is 1 in both. The decimal result is 14.

Assignment operators

Operator	Description	Syntax
=	Simple assignment, assign value on right side to the variable on the left	a = b
+=	Add and assign, add value on right side to the variable on the left	a += b
-=	Subtract and assign, subtract value of right side from the variable on the left	a -= b
*=	Multiply and assign, multiply value of right side with the variable on the left and store to the variable on the left	a *= b
/=	Divide and assign, divide value of variable on the left side by the value on the right and store to the variable on the left	a /= b
%=	Modulo and assign, take the remainder of the division of the variable on the left by the value on the right and store to the variable on the left	a %= b
<<=	Left shift and assign, shift left the value of the variable on the left side as dictated by the value of the right and store to the variable on the left	a <<= b

>>=	Right shift and assign, shift right the value of the variable on the left side as dictated by the value of the right and store to the variable on the left	a >>= b
&=	Bitwise AND and assign, Bitwise AND value on right side to the variable on the left and store to the variable on the left	a &= b
^=	Bitwise XOR and assign, Bitwise XOR value on right side to the variable on the left and store to the variable on the left	a ^= b
|=	Bitwise OR and assign, Bitwise OR value on right side to the variable on the left and store to the variable on the left	a |= b

Misc operators

Operator	Syntax	Description
sizeof	sizeof(a)	Returns the size of the variable
Conditional (? :)	Condition ? a : b	If condition is TRUE returns a, otherwise b
Comma ,	a = b , c	Calculates b and ignores result, returns value of last expression, so a will take the value of c
(.) and (->)	a.b or a->b	Member access operators for structs, unions and classes
cast	type_cast(a)	Casting operators convert data to different types
(&) pointer	&a	Pointer operator (&) returns the address of variable
(*) dereference	*a	Pointer operator(*) points to the variable

Operator precedence

As mentioned earlier operators are evaluated according to well defined precedence. For example, a=7+3*2 will result 42 and not 20, because multiplication has higher precedence over addition and is executed first. If we write a=(7+3)*2, then the result is 20 because parentheses have higher precedence and the addition in them executes before the multiplication.

Here is the list with operator precedence as defined in C++. Associativity is the order in which the operators are executed when more than one is in an expression like a+b+c+d.

In the table below precedence decreases as we go down. Remember that operators with higher precedence, so closer to the top of the list, will be evaluated first.

Type	Operator(s)	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- cast * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right

Shift	<< >>	Left to right
Relational	< > <= =>	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<=&=^=|=	Right to left
Comma	,	Left to right

Statements

Statement in programming, is a complete instruction to do something. We start with simple instructions that usually extend to one line of code. Here are some examples of simple statements:

```
int i;
double d = 0.45;
sum = a + b + c;
diff = d - e;
```

We can group together statements to form complex ones using *curly brackets* like this:

```
{ statement_1; statement_2; ... statement_n; }
```

As you can understand we can nest as many statements as we need to solve our problems.

Every *simple statement* must be terminated with a semicolon. This does not apply to the more complex statements that are enclosed in the curly brackets. The language syntax allows us to write more than one statement in one line of code. It is not recommended though because it may lead to unreadable code. Take a look at the code below and you will see what I mean:

```
int i; double d = 0.45;sum = a + b + c; diff = d - e;
```

I am sure that with minimum effort anyone can make it even harder to read.

Functions

Functions are blocks of code that run only when they are called. We can pass them data referred to as parameters. They are used to perform certain operations such as counting

the words in a sentence or calculating the square root of a number. They are extremely important in code reusability.

C++ code must be within functions. The definition of the language does not support code scattered in the source files but only organized in functions. Every C++ has at least one function. That function must be called *main,* and it is the starting or entry point to our program.

Defining functions in C++ is simple. Here is the form of a C++ function

```
return_type function_name(argument_list)
{
    statements
}
```

A function definition consists of two things. The function header and the function code.

The function header tells us we need to know if we want to use the function. First, we see the return type of the function. Functions in C++ can return a value so their return type can be anything we can store in a variable or *void* if the function does not return anything. Then comes the name of the function. We use this name to call the function when we need it. At the end we have the list with the arguments the function expects. It is a comma separated list of variable declarations. This list can be empty.

The function code is made up from valid C++ statements. From what we have seen so far valid statements in C++ are variable declarations, assignment of values to variables and function calls. As we keep learning more about the language we will see more.

Here is an example of a function in C++

```
double trinomial(double a, double b, double c, double x)
{
    double result = a * x * x + b * x + c;
    return result;
}
```

This function calculates the value of the mathematical function ax^2+bx+c for any values of a, b, c, x. As expected, the return value is of type *double*, and of course it expects four double precision arguments standing for the three constants and the variable. For the name of the function, I have picked 'trinomial.' so I never have any doubts as to what this function does. At the end, the function returns the calculated result. This is done by the *return* keyword. The compiler takes the value of the expression following the *return* keyword, in this case the variable 'result' and returns it to the calling code.

Here is a hypothetical call to the function:

```
double value = trinomial(1.2, 2.3, 3.4, 4.5);
```

The variable 'value' is assigned the value of the function ax^2+bx+c where a=1.2, b=2.3, c=3.4 and x=4.5. Sometimes, depending on the function and what we want to do in our code we ignore the *return value* of a function.

Header files

As the compiler goes through our source files generating code it is evident that it needs to know how to manage everything it encounters. This applies to functions we can call. We must declare them, so the compiler knows their return type and their argument types. C++ is very strict when it comes to variable types.

For this reason, we create some files in which we put declarations of names the compiler might encounter and instruct the compiler to read them using the *#include* directive. When the compiler finds this directive, it stops translating the current file and starts reading the file we tell it to read. At the end it resumes at the line after the *#include*.

We may create a file and call it 'my_math_functions.h' in which we will put the declarations of our custom mathematical library. One of them will be 'trinomial' as follows:

```
double trinomial(double a, double b, double c, double x);
```

We copy the header of the function and put a semi-colon at the enc of the line. When the compiler reads the declarations stores the function information and if it encounters a call to that function, It knows how to manage it.

Namespaces

C++ introduces a way to prevent name conflicts in large projects. Large programs like an internet browser can have several thousand files. The number of the names of all the functions and other globally defined objects, such as user defined types, grows rapidly. It is inevitable that the same name will come up twice creating conflicts.

Namespaces is the method used in C++ to break down and organize in logical groups the global symbol names. This is evident in the standard library. It is in the *std namespace*. This ensures that we can use it in parallel with another library that performs I/O with a specific device and be sure that under no circumstances will we encounter the same name and have a problem.

The syntax of the *namespace* is simple. All we must do is use the *namespace* keyword followed by the name we want to use and enclose everything in curly brackets.

In the header file where we declare our code we write:

```
namespace A { declarations }
```

And in the source file we write:

```
namespace A {
    return_type func_name(arg_type arg)
    {
        statements
    }
}
```

Accessing the functions within a namespace requires us to precede the function name with the name of the *namespace* and the *scope operator '::'*. Here is the call to our 'trinomial' function which can be part of the gdd (*g*ame *d*evelopment *d*emystified) namespace:

```
double value = gdd::trinomial(a, b, c, x);   // function call
```

Basic I/O

Every program is practically useless unless we can interact with it. We need to be able to tell it what to do and get back answer. In technical terms we must do some Input/Output. The basic I/O is direct interaction with the user. Reading input from the keyboard and printing output on the screen.

The standard library has a very flexible and easy to use mechanism to do it. This is the IOSTREAM library. As we mentioned earlier C++ does not provide I/O but relies on the use of external libraries, with the standard library being our first choice among them.

IOSTREAM has two mechanisms, among others, which are used to perform the I/O operations we need. They are *cout* and *cin*. *cout* is used to print on the screen, and *cin* to read from the keyboard. Using the C++ terminology, they are objects that are part of the *std* namespace. Here is an example use of the functionality:

```
std::cout << "insert a:";   // give the user a hint
std::cin >> a;              // read the value of a from the keyboard
```

Here we introduce the streaming operators. They are the *stream insertion* "<<" operator and the *stream extraction* ">>" operator. As the name denotes, the *insertion* operator takes the value on the right and inserts it into the object on the left. The other operator

takes data from the object on the left and passes it to the object on the right. In our example we take a text string and pass it to the object responsible for sending it to the terminal. The second line reads the keyboard until we press the ENTER key and then passes the input to the variable 'a.'

Escape characters

In the beginning of our sample program, we print the classic "hello world" message. At the end of the text though you may notice something strange. A strange looking character sequence appears which is never on the screen. The two characters that we never see are "\n". What are these characters and what is their role?

These are *Escape Characters*. They are used to perform special tasks. The sequence we see here tells the output mechanism to go to the beginning of the next line on the screen. The compiler substitutes these characters with special values which guide the output stream to perform some special actions.

Escape Characters are also used when we want to print the double and single quotation marks, because these characters are reserved for the representation of string and character literals.

Here is a list of the most used *Escape Characters*.

Escape sequence	Action / character represented
\a	Alert Beep
\b	Backspace
\n	New line, go to the start of the next line
\r	Carriage return, go to the start of the current line
\\	Print the backslash (\) character
\'	Print single quote
\"	Print double quote

Our first program

So far we have covered the basics. Now is the time to put them together and create our first program.

Start Visual Studio and create a new project selecting the 'Console App' template. When prompted for a name I recommend you call it 'first_app.' This will make it easier for you to follow the sample code and associate it with the book. Remember ALL the code in this book can be downloaded from GitHub. The console application is a program that runs in the Windows Command Prompt. This makes programming easier because we do not need complex code to deal with Windows and its Graphical User Interface.

Visual Studio created the file ‘first_app.cpp.’ We change the code in it with this:

```
// first_app.cpp : This file contains the 'main' function.
// Program execution begins and ends there.

#include <iostream>

#include "my_math_functions.h"

int main()
{
    std::cout << "Hello World!\n";  // the 'standard' greeting from our first
program

    double a, b, c, x;          // here we define some variables

    std::cout << "This program calculates the trinomial a*x*x + b*x + c\n";
```

```
    std::cout << "insert a:";   // give the user a hint
    std::cin >> a;              // read the value of a from the keyboard
    std::cout << "insert b:";   // another hint for the user
    std::cin >> b;              // now read the value of b
    std::cout << "insert c:";   // another hint for the user
    std::cin >> c;              // now read the value of c
    std::cout << "insert x:";   // another hint for the user
    std::cin >> x;              // now read the value of x

    double value = gdd::trinomial(a, b, c, x);  // call our function

    // output the result
    std::cout << a << "*" << x << "*" << x << " + " << b << "*" << x << " + "
<< c << " = " << value << "\n";

    return 0;
}

namespace gdd {
  double trinomial(double a, double b, double c, double x)
  {
    double result = a * x * x + b * x + c;
    return result;
  }
}
```

Next we create a file called ‘my_math_functions.h’ to put the declaration of our ‘trinomial’ function. When everything is complete press Ctrl+F5 and Visual Studio will compile and run our program.

A thorough look at the code will reveal all the features we have talked about so far.

In the beginning we *#include* the IOSTREAM library and then our own math functions declarations. The first function of our program is the obligatory *main* function, followed by the *trinomial* function inside the *gdd namespace*. I could not resist the temptation and included the classic “hello world” prompt. It would not be a proper first program without it anyway.

The only thing we see for first time is the display of the result before the *main* function returns. There we have catenated many variables and text strings to create a nice and descriptive output to the user. This can be done because the return value of the *stream operator* is the stream object itself. In this case the *std::cout* object which can be used to output the next chunk of information until we are done.

Summary

- C++ is a compiled language
- C++ is very light
- Everything comes in the form of external libraries
- Code is organized in functions which can take arguments and return values

Chapter 2: Flow Control

By *flow control* we mean the order in which out code is executed. Calculating the sum of three numbers is a sequential task. Stepping up to a generalization of the task for a variable set of numbers is a repetitive job, until we reach the end of our set. We must decide whether we are at the end or not in order to repeat the process and add the next number.

The most obvious need to change the flow of our code is when something is not as expected. We have to make this kind of decisions all the time. The car is not the color we want, the weather is not so good for swimming, you name it.

In general, we check the conditions, and we modify our behavior accordingly. The exact same thing arises in programming. Two objects in our game collide so we have to do something. The user pressed the accelerator pedal, so the car has to speed up.

First we must see the *conditionals*. The logical expressions upon which we will base the decision-making mechanism and the programming approach.

With the term *conditionals*, we mean expressions that can be evaluated to *true* of *false*. By convention, apart from *Boolean expressions*, everything that is not zero is considered *true*, while 0 is considered *false*. Here is a list of *conditional operations*:

Conditional	Description
a>b, a<b, a==b, a!=b, a>=b, a<=b	Comparison operations
a && b, a || b, !a	Boolean operations
a, call(), a & b, a | b, a ^ b, a + b…	Actually, everything that can give a valid result

if … else

What should we do if a car is overloaded? We are creating a game involving trucks. The user has put too much stuff on the truck, and it cannot move. How should we write the code?

Here we are using the *if* statement. We are checking the condition and if it is true we execute a piece of our code. The syntax of the code is as follows:

```
if (condition)
    statement
statement(s)
```

We must enclose the condition in parentheses and our code in curly brackets. If the condition evaluates to true or not zero, the block of code within the curly brackets is

executed. Otherwise, the flow jumps to *more_code*. If the condition is met and we enter the block, we are not forced to continue to *more_code*.

There are times when we want to execute a piece of code if the condition is not met before proceeding with the rest of the code. In that case we use the *else* statement.

```
if (condition)
    statement
else
    statement
statement(s)
```

Now our code will execute the *if_code* block if the *condition* is true, or the *else_code* if the *condition* is false. In either case it will continue to *more_code* afterwards. Here is a little example with the truck load.

```
if (weight > 100)
    std::cout << "the truck is overloaded\n";
else
    std::cout << "the truck is correctly loaded\n";
std::cout << "load check done...\n";
```

switch-case

There are cases when we have to take many different paths depending on the value of a variable. Then we can do this

```
if (weight == 1)
{
    // ...
}
else if (weight == 2)
{
    // ...
}
else if (weight == 3) ...
```

This code is perfectly fine. Every time the condition fails the code jumps to the next *else* statement, where we have a check for another condition. This can be hard for any programmer that has to maintain the code. For situations like this C++ has a better solution. The *switch-case* construct. Here is its syntax:

```
switch (/* some integer expression */)
{
case value_1:
    // ...
    break;
case value_2:
    // ...
    break;
...
default:
    // ...
    break;
}
```

The condition in the beginning must evaluate to an integer value. Then this value is checked against the values in the *case* constructs. If it is found to be equal to any of them, the code in that block is executed. If all checks fail the code executes at the *default* condition. The *default* condition is optional.

The break command at the end of each block transfers control to the code after the *switch-case* construct.

There are cases when we want code to execute the next *case* along with the current. Say when *value_1* is met we want to perform some tasks and then perform whatever we have for *value_2*. In this case we can omit the *break* command at the end of *case value_1* block. This will allow the code to execute and break to the end after the *value_2* block.

for

Assume we need to calculate the sum of the numbers between a lower and an upper limit. This job requires a repetition mechanism in the language that will allow us to repeat a piece of code until we reach a desired result.

The first *looping* construct in C++ is the *for* looping construct. This defines a block of code which is executed repeatedly until a terminating condition is met. Here is its syntax:

```
for (init; condition; step)
    statement
```

It is the *for* keyword that marks the presence of the construct. In the parentheses we see three statements. The *init* statement is executed and then the *condition* is checked. If the condition fails the whole block is ignored and the code continues after it. If it succeeds the *code* is executed. At the end, the *step* code is executed, and control goes back to the check of the condition and the circle repeats.

It is possible that the *code* in the loop may never be executed since the *condition* may fail. The *init* code though is always executed.

This sample calculates the sum of some numbers:

```
int sum = 0;
for (int i = lower_limit; i <= upper_limit; ++i)
{
    sum += i;
}
```

We start counting from the lower limit and increment the counter until the upper limit. At every step we add the counter to the sum.

The three statements in the parentheses are optional. Only the semicolons are mandatory. In fact, it is common for loops we want to run forever to write

```
for ( ; ; )
    statement
```

This is usually done in programs that wait for user input to perform a task, like a driving simulator:

```
for ( ; ; )
{
    int steer = steering_wheel_position();
    steer_car(steer);
}
```

This is a very simplified example, but the idea is clear. We are repeatedly reding the position of the steering wheel and pass it to our artificial car.

while

This looping mechanism is almost identical to the *for* loop. Actually, it is a *for* loop without the *init* and *step* statements, but only with *condition*. The syntax of the *while* loop is:

```
while ( condition )
    statement
```

In the case of the *while* loop *condition* is mandatory and cannot be omitted. This is equivalent with

```
for ( ; condition ; )
    statement
```

Here is a simple example:

```
while (it_is_raining())
{
    hold_an_umbrella();
}
```

do … while

C++ has more looping options than the *for* loop described above. The next we are going to examine is the *do … while* loop. This defines a block of code which is executed until a condition is broken. Contrary to the *for* and *while* loops above, the condition is checked at the end of the block. This means that the block of code is executed at least once.

The syntax of this looping construct is like this

```
do
    statement
while (condition);

Or for more clarity if the statement is simple

do
{
    statement
} while (condition);
```

Here is a simple example

```
do
{
    keep_playing();
} while (!quit());
```

break and continue

Now we will see two keywords that allow us to modify the execution of a loop. The first exits the loop and the other transfers control to the beginning of the loop.

We saw *break* earlier when we talked about the *switch-case* branching mechanism. When the program execution reaches the *break* keyword, the control is transferred to the first statement after the loop. This command lets you break the loop at any time. The *break* command usually follows an *if* statement, where we check whether we should proceed with the iterative process or terminate it.

```
for ( ; ; )
{
    statement(s)
    if (we_should_break)
    {
        break;
    }
    statement(s)
}
code_comes_here_after_break
```

So now we can approach an interactive program and its main loop like this:

```
for ( ; ; )
{
    char key = get_keyboard_input();
    if (key == 'q')
        break;
    // perform other tasks based on user input
}
```

And this brings us to *continue*. This command appears within a loop and lets us skip whatever follows it and jump to the beginning of the loop.

```
for ( ; ; )
{
    statement(s)
    if (we_should_skip)
    {
        continue;
    }
    statement(s)
}
```

These *unconditional jumps* as they are also called, are usually placed in an *if* or other branching statement, to control their behavior. As we said if *we_should_skip* the statements that follow the *if* command will not be executed and the loop will start over.

```
for (int i = 0; i < 10; ++i)
{
    std::cout << i << "\n";
    if (i > 6)
        break;
    if (i>3)
        continue;
    std::cout << "hohoho\n";
}
```

In this example we have combined both *break* and *continue*. It will print “hohoho” while *i* is less than 4, and when it reaches 7 it will exit the loop altogether.

goto

This is the simplest and fastest, in terms of machine code and branching mechanism of the language. It transfers control to another part of the code by executing a simple *jump* instruction on the CPU. The point we transfer control to is marked with a label. The same label we use at the *goto* command:

```
label:
// code
goto label;

or

goto label;
// code
label:
```

The *label* can be either before the *goto* command, in which case we create a loop, or after, so we skip some code.

There is an unwritten law in computer programming urging us to avoid using *goto* and rely on other methods to control the flow of our program. Although *goto* is a very simple command, it tends to make code unreadable for humans. This example shows how we can replace a *for* loop with *if* and *goto*.

```
for (init; condition; step)
{
    code;
}

can be rewritten as follows

init;  // perform initialization statement
start_of_loop:  // mark the beginning of the loop
if (!condition) // check condition
    goto end_of_loop; // if it fails (!condition), abort loop

code;  // execute the code in the loop

step;   // perform the step
goto start_of_loop;  // and go back to the start
end_of_loop:
```

The *goto* version of the loop is the actual code generated by the compiler, but the *for* version is the one we have to maintain.

The *goto* command is a leftover from *assembly* code. *Machine code*, as we also call it, does not have all these versatile constructs like loops and conditional execution clauses we can build with modern languages. Instead it has simple comparisons with ZERO and JUMPS. All these high-level features help us express and convert complex program logic into code.

Furthermore they make the code easier to understand and maintain. The above example clearly shows how unreadable our code can become with the use of *goto*s, compared to the simple *for* loop it replaces. If you cannot go without the *goto* you have to rethink your program logic and redesign it.

return

This command is used to return the flow of execution from a function to the line that we placed the call. Apart from returning execution control it can also return data. In C++ we have functions, and functions usually return values. The reason we are putting *return* along with the other glow control mechanisms is because we can use it at any point within a function, to abort it. Instead of using a *goto* command that will take us to the bottom of a function and *return* from there, we can *return* from anywhere in our code.

```
return_type some_function(variable_type some_argment)
{
    statement(s);

    if (enough_processed)
        return value;

    statement(s);

    return another_value;
}
```

Summary

- We can take different paths in code execution
- We can repeat a process until we reach a certain goal
- We can always jump to the desired location in the code

Chapter 3: Arrays

There are times when we need to hold a few similar things in contiguous memory. The coordinates of a point in 3D space for example are three floating point numbers that we need to keep together. Declaring three different variables for each coordinate is not so convenient. It is more convenient to have one variable for our point that can hold the three different coordinates.

Definition of arrays

So far we have seen variables that can be used to store one value only. That may be an integer, a floating-point number, or a simple character. These variables are called *simple variables*. Arrays are variables that can hold more values of the same type, like the three coordinates of a point we saw earlier. These are called *array variables*.

Individual array elements can be accessed via the variable we assigned and an indexing mechanism.

Since arrays are variables we start with their declaration

```
variable_type array_name[array_size]  or
variable_type array_name[array_size] {initial_values }
variable_type array_name[] {initial_values }
```

In the first declaration the three coordinates are *uninitialized* and may have random values depending on the state of the computer memory. In the second declaration we have control over the values because we give initial values to all the elements of the array. The third declaration goes one step further. The compiler deduces the size of the array by the initializer given. It is the most flexible since with one line of code we allocate and initialize an array. We do not have to update the size int the brackets if we add an initialization value.

If we want to define an array to hold the coordinates of a point we would write

```
double coords[3];  or
double coords[3] {1, 2, 3};
double coords[] {1, 2, 3};
```

We create a variable named ‘coords.’ that will hold 3 values of type *double*. As we mentioned before the second declaration assigns the values 1, 2 and 3 to the three elements.

As a rule of thumb always initialize variables when you declare them. However careful you may be, none can guarantee that in a future fix mistake cannot happen with uninitialized variables, and these mistakes are among the hardest to find.

If we want to set the 'x' coordinate, which we assume is first, we write

```
coords[0] = 1.2;
```

Reading the 'y' coordinate which may be at the second position we write

```
double y = coords[1];
```

This is the right time to point out something very important about the C++ *arrays*. The first element of an array is at index **0**. For an array of size *N,* the index is from 0 to *N*-1. This convention may seem odd but is very useful when we use *pointers*, as we will see in the next chapter.

Multidimensional arrays

Icons are widely used to represent anything visually today. Computers, tablets, telephones, and any other device that uses some sort of Graphical User Interface use icons.

Computer images can be imagined as a canvas divided in small square tiles, organized in rows and columns. Icons are small computer images. A common size for images, especially in applications where screen size is small, is 16 x 16 pixels. In the picture below you can see a sample 16 x 16 icon.

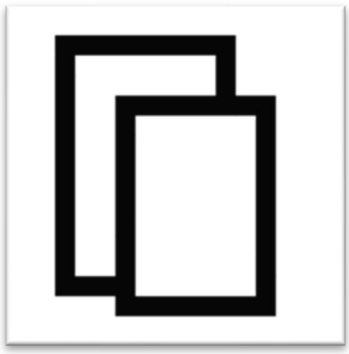

Figure 1: Sample icon

This image is made up of tiles arranged in rows and columns. Some of them are white and some of them are black. If we represent the white tiles with 1 and the black with zero we can create a matrix like this

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1
1	1	0	1	1	0	0	0	0	0	0	0	0	1	1	1
1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1
1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1
1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1
1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1
1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1
1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1
1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1
1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1
1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1
1	1	0	0	0	0	1	1	1	1	1	1	0	1	1	1
1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

We can store this image in an array of 256 integers, but it is not very convenient. We have two-dimensional information so it would be nice to have some two-dimensional storage. C++ supports *Multidimensional Arrays*, so we have what we need.

The declaration of a *multidimensional array* is like this

```
variable_type array_name[rows][columns]  or
variable_type array_name[rows][columns] { initial_values } or
variable_type array_name[rows][columns] { {row[0] values}, {row[1] values} … }
```

Although the second and third declarations are practically the same, the third is better for code maintenance. The compiler can manage anything that conforms to the language standards. The code on the other hand is maintained by humans and they should be our main concern.

Here is how it works. In the example above we need an array of 16 rows, and each row will have 16 columns. We write it like this

```
int img[16][16];
```

This creates an array of 256 integers, only this time they are arranged in 16 'groups' of 16 elements, the *rows,* and the *columns* of the array, like this *{ {e, ..., e}, … , {e, ..., e} }.*

We will use a smaller array of three rows and four columns to illustrate initialization and indexing

```
int img[3][4];
int img[3][4] { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
```

	Column 1	Column 2	Column 3	Column 4
Row 1	img[0][0]	img[0][1]	img[0][2]	img[0][3]
Row 2	img[1][0]	img[1][1]	img[1][2]	img[1][3]
Row 3	img[2][0]	img[2][1]	img[2][2]	img[2][3]

To hold a short animation, we need several images taken within few milliseconds one from the other. A sequence of 10 images showing the icon above transforming into another icon is a great candidate for our next programming approach. Again, we need one variable to hold all the frames. Now we know what to expect, a three-dimensional array.

This is how to declare this time

```
variable_type array_name[arrays][rows][columns]  or
variable_type array_name[arrays][rows][columns] { initial_values } or
variable_type array_name[arrays][rows][columns] { { {row[0] values} … }, …}
```

As we see the three-dimensional array is like the two-dimensional array. All we did was add the number of arrays we want. Going on like this we can add as many dimensions as we need.

Here is a little sample

```
int __animation[2][3][4];
int _animation[2][3][4]{ 1,2,3,4, 5,6,7,8, 9,10,11,12, 1,2,3,4, 5,6,7,8,
9,10,11,12 };
int animation[2][3][4]{ { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} }, { {1,2,3,4},
{5,6,7,8}, {9,10,11,12} } };
```

The more dimensions we add the more complicated it looks. It looks complicated, but it is not. Still, it is difficult to manage, and mistakes can be hard to find and fix. Arrays are a very powerful tool, but thing can easily get out of hand.

Strings

With the term *string* in C and in C++ we mean a collection of characters. Here we will deal with the C variant of the strings. C++ has an object-oriented approach that we will examine later in the book.

Strings in C are *arrays of characters* terminated with the *null* character, which is denoted with '\0'. The actual value as defined by the ASCII standard is 0.

First we look at how to declare a string variable

```
const int string_length = 50;
char variable_name[string_length];
char _variable_name[string_length] = "string_initializer";
char __variable_name[] = "string_initializer";
char ___variable_name[string_length]{ "string_initializer" };
char ____variable_name[]{ "string_initializer" };
```

The declaration is almost like the arrays we have seen so far. The first difference is that we enclose the initialization characters in double quotes. The other difference is hidden but we must always keep it in mind. Strings are terminated with a null character which means that the space they occupy in memory is one more than the characters we see and count.

You can access the characters in a string like the elements of an array.

```
char s[20]{ 0 };
s[0] = 'C';
s[1] = '+';
s[2] = '+';
std::cout << s << "\n";
```

Arrays as arguments to functions

Arrays can be used as arguments to functions just like any other variable type. They can also be returned from functions. Here we are going to see how to use arrays as arguments and accept them as return values. The syntax for passing arrays as arguments is:

```
// function declaration
return_type func_name(type name[length]);

// function definition
return_type func_name(type name[length])
{
    code
}
```

This is how to implement a function that calculates the average of five numbers

```
double calc_average(double numbers[5]);

double calc_average(double numbers[5])
{
    double average = 0;
    for (auto i = 0; i < 5; ++i)
        average += numbers[i];
    average /= 5;
    return average;
}
```

Calling the function is not complicated either.

```
int main()
{
    double numbers[]{ 1,2,3,4,6 };
    double average = calc_average(numbers);
    std::cout << "the average is :" << average << "\n";
    return 0;
}
```

We just have to print the variable name without the brackets. The details are managed by the compiler.

Strings can be passed as arguments in the same way. Here is an example.

```
// count the length of a string
int count_length(char s[]);
int count_length(char s[])
{
    int i = 0;
    while (s[i] != '\0') // until the end of the string
    {
        ++i;
    }
    return i;
}

int main3()
{
    char str[] = "this is a string we want to test";
    int len = count_length(str);
    std::cout << "length of \"" << str <<"\" is:" << len << "\n";

    return 0;
}
```

We have actually made a very simple implementation of the library function *strlen*, which counts the length of a character string. The good thing about this example is that we can demonstrate the *null terminating character* of the string and the use of *escape characters* when displaying the result.

Returning arrays from a function is a little more complicated and we will cover it in the next chapter where we cover pointers.

Summary

- Arrays can be used to store values of the same type
- C++ supports multidimensional arrays
- Character strings are arrays of characters
- Arrays can be used as arguments to functions

Chapter 4: Functions

In the first chapter we saw that functions are logical blocks of code that do a specific task. We covered all the basics so we can start organizing our code in functions. In this chapter we will take an in-depth look at functions and learn those features that will help us create efficient and robust code.

Functions

A function is a block of command that when executed they perform a specific task. We use them to organize our code into reusable blocks that we call any time we want to perform a clearly defined task. For example, we might create a function that calculates the acceleration of an object if we know its mass and the sum of the forces that act upon it.

Let us focus on the declaration of a function for a while.

```
return_type function_name(argument_list);
```

It describes the way the function communicates with the rest of the program. First is the *return type*, what we should expect to get as a response. In the example above is should return the acceleration of the object.

Then we have the name of the function that we can use to call it.

Finally, we have the list of arguments. They are used to specialize the function for a specific job. In our example they should be the mass of the object and, the sum of the forces. So, each time we call it we 'specialize' it with the parameters for a specific object.

The argument list is a very important characteristic of a function.

Static functions

Functions can be called not only from within the source file they are defined but from any source file in our project. In the first chapter we described the use of *header files*, and the declaration of functions. That is a nice feature when we want to share the code. There are cases though that we want to keep implementation details hidden from the rest of the project.

One example is the network communication between players in a multiplayer game. All our game needs are the commands the remote player has issued and a status update. Another example is the code that takes user input. It does not make any difference if the user prefers the keyboard or the mouse or any other device, or even how he configures

them. We just create a hidden code layer that does all the work for the network or user input and our program can only get the information it needs without bothering for what goes on behind the scenes.

```
// static function declaration
static bool open_network_connection();
// static function definition
static bool open_network_connection()
{
    return false;
}
```

The functions we want to keep hidden are not declared in a header and are preceded with the keyword *static*. We can declare them at the top of the source file always using *static*. You can even create functions with the same name in other files to manage similar tasks and be sure that they will never be an unresolved conflict for the compiler or the linker.

Function overloading

In C++ we have a feature call *function overloading*. This means that we can have two or more functions with the same name. The compiler relies on the argument list and the strict type of system of the language to determine which function to call each time. As long as the functions have different types or number of arguments they can have the same name. Here is an example of functions with different argument lists

```
void fancy_printer(int i);
void fancy_printer(double d);
void fancy_printer(char* c);
void fancy_printer(int i, char* c);
```

These functions can coexist but how does the compiler decide which function call to use?

The definition of the language states clearly the type of the argument we use in the call determine the function we call. In this example if we pass an integer value the first function is called, if we pass a double precision number the second and so on.

```
fancy_printer(1);         // calls the first
double d = 2.75;
fancy_printer(d);         // calls the second
char str[] = "abcd";
fancy_printer(str);       // calls the third
fancy_printer(3, str);    // calls the fourth
```

This feature takes away a great load. We no longer have to remember different function names to do essentially the same thing. These functions are probably doing the same thing, they produce some fancy output. Their only difference is the type of the data they

output with the fourth taking an extra parameter for a richer output. Whatever the reason maybe we can design our program and rely on the compiler to do the trivial tasks.

Default arguments

C++ allows the definition of default arguments or default values for arguments. This means that trailing arguments can be omitted, and the compiler will fill the call with default values we have defined.

The definition of C++ says that if one argument has a default value, then all the subsequent arguments must have default values. We can start omitting arguments from right to left. We cannot omit one argument and continue with the rest.

Here is an example to illustrate *default arguments*.

```
int multiply(int i = 1, int times = 1)
{
    return i * times;
}

// the first call returns the number tripled
m = multiply(2, 3);
// while the first simply returns the number
int m = multiply(2);
// finally, the third returns the absolute default
m = multiply();
```

As expected the first call passes the two values to be multiplied. In the second call the *times* parameter is omitted, and the compiler uses the default value 1. In the third call we leave out both parameters and the compiler fills in the default values.

Many times, we design functions to perform some tasks and on rare occasions differentiate their behavior. For instance, this might be a function calculating the rebound speed of a ball after it hits the ground. This calculation considers the loss of energy of the bounce. In a special calculation though we want to ignore energy loss. Then we must pass a parameter to the function telling it to do so. If the last argument of the function is by default *true*, telling it to do the full calculations, we would end up with a clear code where none wonders about what the argument means, and in the one end only case we want them to be careful the see the passing of *false*.

This leads to code being easier to read and maintain, with less chances for something to go wrong. This is another feature that helps us create more robust code faster and easier.

Function variables

There is more to variables than just their type. The type is just how much space the variable takes and what values it can hold. But where is that space allocated? When is that space no longer available?

In this section we will take a closer look at the variables, their scope and lifespan and the storage they take. What the different types are and how the differ from each other.

Local variables

Local are the variables that declared inside a function. In all the examples we have seen so far the variables are local. They were declared within the body or *scope* of the function. Their so called 'lifespan' is from their declaration to the end of the function. These variables that initialized when their declaration is reached and release when their scope are also called *automatic* variables.

There were variables that were declared in the declaration of a *for* loop. In older definitions of the language their lifespan was until the end of the function. In modern specification their lifespan is until the end of the loop. Compilers enable one of the two via compilation options. I would recommend though that you go with the latest specification.

In general variables 'live' inside the block of code they are declared. With block of code, we mean the statements that are enclosed within curly brackets. So, if you declare a variable inside the block for an *if* clause then it is not accessible outside that block.

The following example shows it all

```
// the arguments are local variables initialized automatically
// with the values we pass in the function call
// notice the pointer! it holds an address not a value
void local_variables(int i, double* d)
{
    int j;

    for (int k = 0; k < 10; ++k)
    {
        std::cout << k << "\n";
    }

    ++k;   // error k is out of scope

    // perfectly valid statement, only the value is NOT returned
    // the variable has this value until the end of the function
    i = 100;

    if (i > 10)
    {
```

```
        double l = 100 * (*d);
    }

    std::cout << l << "\n";  // error l is out of scope
}
```

Now we can show why we need pointers to refer to variables over function calls. Function arguments are actually local variables. The compiler creates their initialization code with the values we put in the call. There is where we place the address instead of the contents when we push a pointer. All local variables are destroyed upon function exit and the arguments as well. Having the pointer though the function can modify variables beyond its small world and so we get back data.

We mentioned earlier the *stack*. This is a contiguous memory block used to store local variables. All local variables are allocated on the program stack. The term stack means that it builds up like stack. Every time a function is called it allocates a space on the stack big enough to hold all the local variables. When it exits that memory is returned to the system.

When this local storage space is allocated it is not initialized automatically so if we do not initialize local variables they end up with random values generated by the state of the memory.

Another limitation is the size of this memory. In modern systems it is quite large but limited anyway. This is the reason we try to avoid statically defined local arrays if they are too big. They take up a lot of our limited space.

Global variables

C++ permits the definition of variables outside of functions. Those are called *global* variables. They are accessible from anywhere in our code. Every function declared after them can read and write to them. That is because the compiler needs everything to be declared before being used.

Global variables are allocated before the *main* function is called and live throughout the runtime of the program. Their usage is not limited to one source file only but, all can have access to them. If we need to access a global variable in a file other that the one it was declared in, we use the keyword ‘extern.’

The following example will make it all clear:

```
// in a source file called my_globals.cpp
int some_global_int = 0;

// in a source file called my_code.cpp
extern int some_global_int;
```

Notice that we can have only one initialization for our global variable. Is better to have the initialization with the declaration. In any other case we will lose a lot of time trying to find it if we need to do so.

We have seen the *heap* is the part of the computer memory that is used for dynamic memory allocation. In the beginning of this memory the compiler reserves some space for the needs of the program. One of the reserved chunks of memory is reserved for the global variables. This memory is reserved and organized before the program starts executing.

This memory space has a lifetime equal to the lifetime of the program and is accessible from anywhere in the program hence the name 'global' for the variables stored there.

Static variables

Some variables are declared 'static'. If you go through the code of a big project you will definitely come across the *static* keyword before a variable declaration. This might appear before a global variable or a local one.

Static global variables

As we saw with static functions earlier in this chapter, the *static* keyword prevents the variable from being used in the code from other source files.

In general, it is a good practice to keep task implementation specific information hidden from the rest of our code. Having s*tatic* data means that we can revisit and modify our code, for any reason we might want to do it, and the rest of the program is affected. The new and improved version will integrate seamlessly, and we will have less things to test.

Static local variables

Static variables can exist in a function as well. The lifespan of these variables is not limited to the scope of the code they are declared but, instead they live until the end of the program. They are initialized the first time they are encountered, and they can be accessed every time retaining their value.

They are statically allocated in the heap like global variables and not on the stack like local. This means that there is only one instance of them accessible only in the scope they are declared. This means that like local variables, if the static variable is declared inside the scope of an *if* clause, it is not accessible outside of it, even though it still exists.

One possible use of static variable is a counter of how many times a function was called, or as a flag to prevent a function from running more than a limited number of times.

Here is the code:

```
// count how many times we run
int run_counter()
{
    // start with initialization
    static int count = 0;
    // count
    ++count;
    // and return current status
    return count;
}

// this function will run only once
void function_runs_once()
{
    // initialize a static variable to 0
    static int i = 0;
    // if the value of it is 1 return
    // and ignore the rest of the code
    if (i == 1)
        return;
    // set the value to 1
    i = 1;
    // the code from this point on runs only once
}
```

We put the initialization code in the declaration so that it runs only once. The rest of the statements in the function run every time and an assignment that was supposed to be initialization will ruin everything.

Recursion

Recursion occurs when a function calls itself. Either directly or indirectly by calling another function which ends up calling the first again.

```
// indirect recursion
// one function is calling the other
void recur_i();
void secondary()
{
    recur_i();
}
void recur_i()
{
    secondary();
}

// direct recursion
// function calling itself
```

```
void recur()
{
    recur();
}
```

If you take a look at the examples above you will notice that these functions never return. The loop forever. This is a common mistake when we implement a recursive function to solve a problem. We must provide a path in our code that returns without recursion and the whole thing unwinds.

If we fail to do so, our program will crash due to *stack overflow*. We saw earlier that the local variables are allocated on the stack each time the function runs. This means that every call has its own copies of the variables. Running a function recursively we take more and more stack space, and if we take up all available space the program crashes.

Here we will see the classic problem of calculating the factorial of a number. As we know from mathematics the factorial of a positive integer *n* is the product of all integers from 1 to the number *n*. It is denoted n! and is

n! = n*(n-1)*(n-2)*…*2*1

the code for this would be like this

```
int factorial(int n)
{
    int f = 1;
    for (auto i = 1; i <= n; ++i)
        f *= i;
    return f;
}
```

A closer look to the mathematical definition gives us this

(n-1)*(n-2)*…*2*1 = (n-1)!

And if we substitute we get

n! = n*(n-1)!

A factorial is defined using another factorial. And this is a recursive definition of the factorial function. Here is the code for this new implementation

```
// recursive implementation of the factorial
int recursive_factorial(int n)
{
    // according to the mathematical definition
    if (n <= 1)
        return 1;

    // proceed recursively
    return n * recursive_factorial(n - 1);
}
```

This code implements the function above. It is critical to have the mathematical limitation that will break the recursion when we reach 1, or we invoke the function with 0 or negative number. The correct mathematical implementation never fails. The function is well defined in math.

Returning a reference

Functions can return a reference to a variable. This way the calling function can read the value of the variable, and also set its value. This feature allows the function to appear on the left side of an assignment operation.

Take a look at the code below

```
// a static array somewhere in our code
static int myvalues[10];
// the only way to set the values in the array
// is by using this function
int& getValue(int index)
{
    // if the index is within the array
    // return the appropriate element
    if (index >= 0 && index < 10)
        return myvalues[index];
    // otherwise return a dummy variable
    // the variable must be static
    static int dummy = 0;
    return dummy;
}
```

The function returns references to array elements. We use the array to perform some tasks and it is somewhere in our code. We declared it static so we can hide it from the rest of the program. So, if we want to set an element we go through the ‘getValue’ function. If the index is correct the function returns the corresponding element, otherwise it returns a reference to a ‘dummy’ variable.

This technique hides the implementation details while ensuring that the assignment will never fail. We can change the size of the array at any time and if we modify the function accordingly the program will continue to run without any problem.

The only note is the 'dummy' variable. It must be static or global. We cannot return a reference to a local variable that will be invalid after the function exits.

Pointers to functions

Pointers in C++ can point to functions. Functions are stored in computer memory, so they have a physical address. Storing these addresses allows us to call them using pointers. Function pointers can point to different functions at different times during the execution of the program. This feature enables the dynamic change of the program function.

This is extremely useful when we wand to modify the behavior of standard algorithms. Passing different function pointers, we can get different results according to the needs of our program. Take a look at this example before we analyze the details.

```
// function that takes a function pointer as an argument
void run_iteration(void (*iteration_callback)(int))
{
    // do some task
    // and call a user defined function
    // when needed
    for (auto i = 0; i < 10; ++i)
        iteration_callback(i);
}
// user defined callback function that prints even numbers
void my_callback_evens(int v)
{
    // print even numbers
    if ((v % 2) == 0)
        std::cout << v << "\n";
}
// user defined callback function that prints odd numbers
void my_callback_odds(int v)
{
    // print even numbers
    if ((v % 2) == 1)
        std::cout << v << "\n";
}

// and we call it like this
run_iteration(my_callback_evens);
run_iteration(my_callback_odds);
```

This is a simple use of a function pointer. We have a function that runs a simple loop and at every iteration it calls a user provided function. In the 'run_iteration' declaration we see a

rather peculiar argument list. It declares the argument called ‘ iteration_callback’ which is a pointer to a function that does not return anything ant takes an integer as argument. We must enclose the asterisk and the argument name in parentheses to override operator precedence.

The following two functions ‘my_callback_evens’ and ‘my_callback_odds’ are function that conform to the prototype expected by ‘run_iteration’. So, the first call will print the even numbers and the second will print the odd.

We do not have to do anything to get the function pointer. The name of the function is the pointer to it.

Here is the syntax of the pointer to a function.

```
return_type (*argument_name)(arg_type1, arg_type2…)
and the declaration of the function that takes such an argument
return_type1 func_name(return_type (*argument_name)(arg_type1, arg_type2…));
```

We can make the syntax easier to read by defining custom variable types like this

```
typedef return_type (*new_type_name)(arg_type1, arg_type2…);

and now we can declare variables like this
new_type_name myfunptr1;

and declare arguments
return_type1 func_name(new_type_name argument_name);
```

We used the *typedef* keyword to define a new type called ‘new_type_name’ which stands for the function pointer type we want.

The drawing function in a game can become very long and complex. Every time we develop a new level we must update it with the risk of creating bugs. Function pointers can help us develop a clean solution. We can create a different function for the different states of our game and set a global function pointer, pointing to the right function. Then our main game loop can use this pointer calling the right function every time.

```
// first we define the function type
typedef void (*draw_function)();
// and we declare the interface function
void set_draw_function(draw_function function_pointer);

// this is the default draw function
static void draw_idle()
{
}
// we create a pointer and initialize it to the default function
static draw_function draw = draw_idle;

// the interface function
// we call this when we need to change the drawing function
void set_draw_function(draw_function function_pointer)
{
    draw = function_pointer;
}

void main_program_loop()
{
    while (true) {  // loop forever
        // do our program stuff
        if (user_wants_to_exit)  // user selected to terminate
            break;
        // and finally call draw to update the scene
        draw();
    }
}
```

We could have used a long *switch* statement or any other branching. Then we should have to update the code every time and waste more and more time for the program to branch to the right drawing function. Using function pointers, we have a very simple and clean main function. The drawing was split so problems can be found ad fixed easily.

Summary

- In C++ we can use functions to organize our code
- Functions can be overloaded and have default arguments
- The *stack* and the *heap* are used to store *local* and *global* variables
- C++ supports recursion
- Functions can return references to variables
- C++ supports function pointers
- Lambdas

Chapter 5: Pointers

In this chapter we will see one of the features that makes people love or hate C++. This feature is the *pointers*. Over the years I have seen people avoiding C and C++ because of *pointers*, while others are attracted by this feature alone. Once you understand the concept behind them and you start using them, you are holding a very powerful tool that will help you improve your code dramatically. If you do not understand them correctly or you go too far, you may create a mess out of the code.

Pointers

In chapter 1 we have seen the standard type of variables, such as *int, double* and so on. Now we are introducing a new type of variable, the *pointer*. The variables we know so far are used to store values. This type is no exception, only the value it stores has a special meaning. Instead of holding a value directly it stores the memory address where the value is stored.

Every variable has a place in computer memory. That place is fixed, and the first use of pointers is storing addresses of other variables. This gives us the flexibility to alter the content of that memory location and ultimately modify the value of the variable.

Before we try to find how to use *pointers* for our benefit we will see the syntax and simple use of them. When we are comfortable with the syntax we will see the real strength and flexibility they give us.

When declared, *pointer variables* require a type and are preceded with an asterisk. The type tells the compiler what type of variable we are going to point to, and the asterisk is the *pointer* notation.

```
int *ptr;
```

This sample tells the compiler that a variable called ‘ptr’ will be used to point to a location where an integer is stored. The int keyword can be replaced with any variable type we want. The next step is to get the *address* of an integer and store it in ‘ptr’. For this we will use the *address off* operator.

```
int *ptr;
int var;
ptr = &var;
```

The *address off* operator returns the address of the memory location where our variable resides.

Now that our pointer has stored the location of variable ‘var.’ we can use it to modify its value indirectly.

```
*ptr = 10;
```

Which is equivalent to

```
var = 10;
```

The asterisk (*) is the *dereferencing operator*. It tells the compiler to store the value NOT in ‘ptr’ but instead store it at the location ‘ptr’ points to. It supplements the *address of* operator by giving us reading and writing access to the memory location and its contents.

The *dereference operator* allows us to read the contents of the memory as well.

```
int j = *ptr;
```

Which equivalent to

```
int j = var;
```

This time the *dereference operator* means ‘read the contents of the location pointed to by ptr’ and not ‘ptr.’

Assigning a value to a variable or, reading from it is the direct way of accessing it, while via a *pointer* is the indirect.

Here we must point that by declaring a pointer does not mean we can use it to store or retrieve information. We must point to a valid memory location before we start using the pointer. Nasty thing will happen to our program if we use uninitialized pointers since we are accessing random areas of the computer memory we are not allowed to.

Pointers and Arrays

Pointers and *arrays* are closely related in C++. We have seen that *arrays* are values taking up consecutive locations in computer memory. *Pointers* are variables that hold addresses of other variables. It does not take long to realize that the array variable is similar to a pointer. Despite the semantic difference, the array variable holds the address of the first element.

```
int arr[10];
int *ptr;
ptr = arr;
```

This example shows that the variable holding the *array* is actually a *pointer*. We do not need the *address of* operator to get the address now. We are assigning the value of one pointer to the other just like any variable of the same type. This assignment is the same as

```
ptr = &arr[0];
```

The array variable is the address of the first element.

A *pointer* can be used to access all the elements of the array. The location pointed by the *pointer* holds the first element of the array. Advancing the pointer by one and we get to the second element, by two to the third and so on:

```
ptr = &arr[0]
ptr + 1 = &arr[1]
ptr + 2 = &arr[2]
...
```

The *dereference operator* can access the actual values stored in the respective elements:

```
*ptr is identical to arr[0];
*(ptr + 1) is identical to arr[1];
*(ptr + 2) is identical to arr[2];
```

The void pointer

Before we proceed with what we can do with pointers we must see a special pointer. That is the *void* pointer. This pointer has no type. It can still hold an address of a memory location but, we cannot dereference and read form or write to it.

We can only use it for memory operations that do not require a specific type to work but rather treat memory as raw bytes, like copying large memory block from one location to another.

We cannot dereference *void* pointers, if we want to access their contents we must *type cast* them to the desired type and the read from or write to them. *Type casting* is like looking through glass. It allows us to treat a memory location with a different perspective than the one it was declared. Here is how it is done

```
void *vptr;
int *iptr = (int*)vptr;
```

From this point on we can use ‘iptr’ to store and retrieve integer values in memory, or any type we cast the void pointer to.

We will revisit the void pointer in the memory management section later in this chapter.

Pointer arithmetic

Pointers store addresses and addresses are integer values. This means that we can perform some integer arithmetic with the value of a pointer and access different locations. We used some basic pointer arithmetic when we talked about arrays and pointers.

Here I would like to address a misconception I have come across many times. We are dealing with the contents of the pointer and not the location it points to. When we modify the pointer we make it point at a different location. Picture it like a sign on the road. We actually print the name of a different city and turn to point to that city. Neither the previous nor the new city has moved. It may seem trivial, but I have faced this misconception so many times I feel I should point it out.

Pointer arithmetic is available only to pointers that have a type other than *void*. This restriction comes from the fact that the compiler needs to know the size of the elements the pointer points to, because the changes of the pointer are done in element size. As we saw in the arrays section, adding 1 to the pointer moves to the next element. The memory is addressed in bytes, so if the element is a character the pointer will move by the size of a character which is 1 byte. If on the other hand it is a double precision floating point number it will move by 8 bytes. All this is taken care by the compiler, but it is our responsibility to set it up properly.

We are creating a little program to demonstrate the use of pointer arithmetic. This program contains a function which takes a sorted array of integers and counts the number of unique ones while moving them to the front of the array.

```
#include <iostream>

// eliminate duplicate integers in a sorted array of length len
int array_compactor(int* arr, int len)
{
    // we read and write at the same array
    // the reading pointer advances at every iteration
    // while the writing stays while we encounter duplicates

    // we need a pointer to the writing position
    // initially set at the start of the array
    int* wr=arr;
    // advance the reading pointer, we have read the first element
    ++arr;
    // so, the length of the 'clear' array is at least 1
    int l = 1;
    // the reading pointer is at the first position
    // at advance the reading position and the reading counter
    // until we reach the array length
    for (auto i = 1; i < len; ++i, ++arr)
    {
        // if the current element is not the same as the previous
```

```
        // wr points to the previous and arr to the current
        if (*wr != *arr)
        {
            // advance the writing pointer one position
            ++wr;
            // and increment the length of the 'clean' array
            ++l;
        }
        // copy the content of the reading position
        // to the writing position
        *wr = *arr;
    }
    // return the number of unique elements
    return l;
}

int main()
{
    std::cout << "compact array sample\n";

    // create a sorted array with duplicate elements
    int arr[]{ 1,2,2,2,2,3,4,4,5,5,6,7,7,8,9,10 };
    // use the sizeof operator to count the number of elements
    // we divide the size of the memory allocated by the size of the element
    int len = sizeof(arr) / sizeof(int);

    // call our function to 'clean' the array
    // and return the number of unique elements, and reset len
    len = array_compactor(arr, len);
    // after the call the array is NOT shorter
    // we just moved the unique elements in the first len positions

    // print the 'clean' part of the array
    // we could use arr[i] notation but we are using pointer arithmetic
    for (auto i = 0; i < len; ++i)
        std::cout << *(arr+i) <<",";
    std::cout << "\n";

    return 0;
}
```

This sample is a classic test given to applicants for C++ jobs. You can ‘compact’ the array in place, without using another array for temporary storage. It also demonstrates how we can count the elements of an array using the *sizeof* operator.

Pointers and strings

Strings are arrays of characters as we have seen earlier. So, a string variable is a pointer to characters similar to an integer array. The extra functionality that strings have over any other array is that they are NULL terminated.

In general, NULL means zero, and the zero character is noted as '\0'.

This functionality was inherited from C and thus these strings are called C-strings. In fact, C++ offers a better way to handle strings. We will cover the C++ strings later when we talk about the C++ standard library. For now, I would like to give an example of character pointers, to help you get comfortable with the use of pointers and pointer arithmetic.

This next example is another job interview problem. You are asked to find if a string is palindrome. That means if you can read the string from the end to the start as well.

```
// check if the string is palindrome
bool is_palindrome(char* arr)
{
    // setup initial conditions
    // a pointer to the first character
    char* start = arr;
    // and another one to the last
    char* term = arr; // start at the beginning
    while (*term != '\0')  // go to the terminating '\0'
        ++term;
    --term;  // and back up to the last character

    // the term pointer points to higher address
    while (term > start)   // and while this is true
    {
        // if the characters at the opposite sides
        // are not the same, return false
        if (*term != *start)
            return false;
        // otherwise advance the start pointer
        // and back up the term pointer
        // this way we examine symmetrical positions
        // and the pointers will cross at the middle of the string
        // when they cross the loop will break
        ++start;
        --term;
    }
    // reaching this point means no error was found
    // it is safe to return true
    return true;
}
```

Here is how we can use this function:

```
// const means that the string cannot me modified
char str[] = "abcdedcba";
if (is_palindrome(str))
    std::cout << "palindrome\n";
else
    std::cout << "not palindrome\n";
```

This example has it all. Pointers compared, memory contents compared, pointer arithmetic, you name it. I want to point the difference between the comparison in the line `while (term > start)` and the line `if (*term != *start)`. The first compares the pointers themselves, the memory addresses, while the second compares the data stored in those addresses.

Pointers and functions

In this section we will explore a little more the use of pointers as arguments to functions. We encountered this when we talked about passing arrays and strings to the functions. Now we are going to see another practical use of pointers.

There are times when we want a function to modify the values of its argument variables. Suppose we want to read the location of an object in our 3D game. It would be nice if we had a function that could write the x, y, and z coordinates in three variable we pass to it.

We can do it by passing the addresses of our variables instead of the variables themselves. Here is how we do it

```
// this function takes the addresses of the variables
void read_position(double* x, double* y, double* z)
{
    *x = 1;
    *y = 2;
    *z = 3;
}


… And here is the call


// ---------------------------------------
// here we declare our variables
double x, y, z;
// and we call the function passing it their addresses
read_position(&x, &y, &z);
std::cout << "x:" << x << " y:" << y << " z:" << z << "\n";
```

We are using the *address off* operator to pass the addresses and, then we are using the *dereference operator* to read or write to those addresses and ultimately modify the original variables.

This method is called *call by reference* because it passes a *reference* of our variables to the function, allowing it to access and modify them. Compare this with the following

```
// this is a call by value
void show_position(double x, double y, double z)
{
    std::cout << "x:" << x << " y:" << y << " z:" << z << "\n";
}


… and the call to it



// and pass it to another by value
show_position(x, y, z);
```

This is the *call by value* method of passing arguments. In this call the contents of the original values are copied to the function arguments and so any changes will not come back. Behind the scenes the call is always by value, only in the first place the values we pass are the addresses of the variables, which enable us to access the variables themselves.

References

In C++ pointers were inherited from C. They are very powerful, but they can be dangerous. They offer direct memory access, and they can be used as references to variables. Their flexibility is both their strong and weak feature. It is very easy to make a mistake and they are very complicated to many programmers.

C++ offers a safer and easier to understand way to have a variable than can be used to access another. This is the *reference* type. This variable is not a *pointer* that points to a memory location but an alias to a variable. Internally it is implemented as a pointer but, that is handled transparently by the compiler, and we do not have to worry. A reference is a pointer that cannot be made to point to another variable and dereferencing is handled automatically by the compiler.

References are initialized upon declaration and cannot be modified later. Take a look at the code

```
// first we create a variable
int i=10;
// then we take a refence to it
int& ri = i;

// sometime later we create another variable
int j=0;
// this assigns the value of j to where ri refers (i)!
ri = j;
```

The fourth line shows how a reference variable is initialized. That is the one and only time we can make it refer to a variable. From that moment on this relation cannot be modified. The *reference* is locked to accessing the variable and every time we assign a value to it we simply modify the original variable.

References are so useful within a function. Their real power is when they are used as arguments to functions. There they offer the benefit of accessing the original variable with the additional safety of automatic *dereferencing*.

Here is how our previous example will be after we change from pointer to reference

```
void read_position(double& x, double& y, double& z)
{
    x = 1;
    y = 2;
    z = 3;
}

// here is the call
double x, y, z;
read_position(x, y, z);
```

The function is declared to accept *references* instead of *pointers*. In the call we pass our variables. The compiler takes care of everything. It initializes the *references* and does all the dirty job of dereferencing and setting the values. The syntax is simpler, and the automation minimizes the potential mistakes.

Pointers and dynamic memory management

With variable referencing out of the picture we are left with the real power of the pointers in C++. That is *Dynamic Memory Management*. The ability to allocate and release system memory based on our needs at any time.

Before we start talking about memory management, we must have a clear understanding of how computer memory is organized.

The stack

The *stack* holds all the variables in a function. Integers, strings, arrays, and every other variable type we have seen so far are stored in the *stack*. It is a very fast and efficient storage that really adds to the strength and versatility of C++. But there is a drawback. All variables must be statically defined. This is not an issue for variables like integers, but arrays are not very well suited because we cannot expand them if we need

The heap

The unused memory of the computer is called *the heap*. That memory is maintained by the operating system, and we can use it to allocate large chunks for our needs. We can request memory chunks of variable size any time we need and release it back to the system if we no longer need it. This memory is a little slower to access than the *stack* but with the flexibility it offers it is worth the price.

The problem to solve

Not all our memory needs in a program can be predefined. We do not know in advance how many players will connect to our game for example. This makes it impossible to have an array of players with a predefined size. We might be wasting storage space or even worse run out of space. Another example is the small icon we created when we talked about arrays. How can we handle images of variable sizes? All these are problems that make *dynamic memory management* a necessity.

new and delete operators

C++ has two operators designed to access the *heap* and allocate or release memory. These are the *new* and *delete* operators. The *new* operator requests the memory from the system and the *delete* returns unwanted memory back to the system.

Here is the syntax for the *new* operator

```
data_type* var = new data_type;
```

data_type stands for any type of variable, might that be *int*, *double* etc. Replace it with *int* and you get this

```
int* var = new int;
```

The *new* operator returns the pointer allocated by the system. There is no guarantee though the operation will succeed and in that case the return value is NULL. This is a good enough reason to initialize our pointers to NULL and check their values against it to make sure we have a valid pointer to use.

When we no longer need the allocated memory we **must** return it back to the system like this

```
delete var;
```

The delete operator accepts the pointer variable.

The C equivalents for *new* and *delete* are *malloc* and *free*. They are still available, but they are not recommended. The C++ standard mechanism offers a lot more flexibility as we will see later when we cover objects and classes.

Allocating space for simple values is not so special. It just adds to the complexity of the program. *Dynamic memory allocation* makes the difference when we need to handle large memory chunks, like allocating space to hold the vertices of a 3D object in a game.

Here is how we can allocate memory for a user defined number of points. First we allocate a single dimensional array with three values (x, y, z) for each point

```
double* allocate_one_dimensional(int num_points)
{
    // allocate 3 doubles for each point (x,y,z)
    double* coords = new double[num_points * 3];

    // and return the allocated memory
    return coords;
}
```

This is easy to understand. We allocated enough memory to hold three double precision numbers for each point. Notice the indexing when we access the coordinate data moving by three numbers per step and adding 1 and 2 to access y and z.

We can avoid this indexing if we allocate a two-dimensional array

```
double** allocate_two_dimensional(int num_points)
{
    // first allocate a pointer for each point
    double** coords = new double* [num_points];
    // then allocate memory for each point
    for (auto i = 0; i < num_points; ++i)
        coords[i] = new double[3];

    // this method is not efficient and should not be used for such simple
cases
    // it is presented here for educational purposes only

    return coords;
}
```

This time indexing is very easy, but allocation of memory is more complex. First we must allocate an array of pointers that will hold the coordinates for each point and, then we must allocate each point individually. To release the memory, we must do the job in reverse order. The main disadvantage of this approach is not the complexity of the allocation/deallocation of memory but the fragmentation of the system memory. Instead of allocating a contiguous block of memory to store our data we end up with data scattered all over the computer memory, which makes any operation very slow. Especially

in modern computers with multiple cores and threads it is essential to have our data well sorted and in contiguous blocks of memory.

Here is a small function demonstrating the two allocation methods we just presented.

```
void do_stuff_with_points(double* points, int num_points)
{
    for (auto i = 0; i < num_points; ++i)
    {
        // remember each point takes up three places
        double x = points[i * 3];     // x is first
        double y = points[i * 3 + 1]; // y is second
        double z = points[i * 3 + 2]; // z is third
    }
}

void do_stuff_with_points(double** points, int num_points)
{
    for (auto i = 0; i < num_points; ++i)
    {
        // remember each point takes up three places
        double x = points[i][0];     // x is first
        double y = points[i][1]; // y is second
        double z = points[i][2]; // z is third
    }
}

void dynamic_memory_demo()
{
#define NUM_POINTS 100

    double* points = allocate_one_dimensional(NUM_POINTS);
    // do what we want with the points
    do_stuff_with_points(points, NUM_POINTS);
    // always release the memory when we no longer need it
    delete[] points;

    // and now an inefficient way to do it
    double** points2 = allocate_two_dimensional(NUM_POINTS);
    // do what we want with the points
    do_stuff_with_points(points2, NUM_POINTS);
    // always release the memory when we no longer need it
    // this time releasing the memory is not so easy
    // release each point
    for (auto i = 0; i < NUM_POINTS; ++i)
        delete[] points2[i];
    // and finally release the pointers array
    delete[] points2;
}
```

How memory is actually organized

Now that we have covered the pointers and their usage is a good time to take an in-depth look at how the computer memory is organized from the running program point of view.

This will let you understand how memory management actually works, where the various parts of the program resides and how they function during program execution.

When the operating system loads our program the memory is organized as this image shows

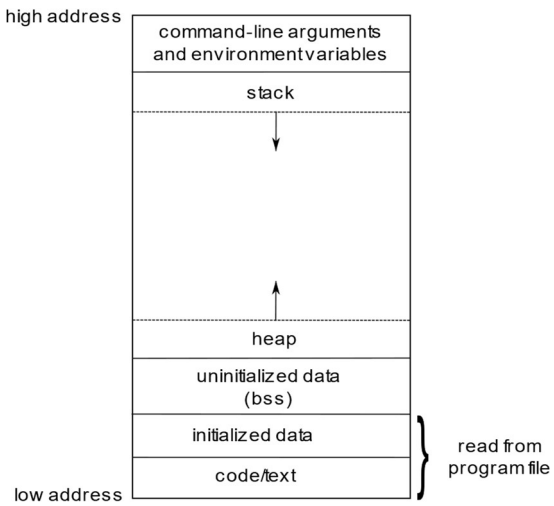

Figure 2 Memory map of a C++ program

These are the main areas of the memory our program occupies.

1. At the lowest memory address we have the *code/text* area. There the operating system loads the machine code instructions the processor executes when running our program. This memory is **read-only**. It is protected by the processor and when any other program tries to modify it the system crashes. This memory can be shared between multiple instances of the program.
2. *Initialized data segment*. This segment contains the initialized **global** and **constant** variables. These are the variables that were declared and initialized in our code. Variables that can be modified are placed in the read-write area while variables declared as **const** are placed in the read-only area.
3. *Uninitialized data segment*. Here are stored all the global variables that were not initialized in the code. Some compilers initialize all the variables in this segment to 0, but there is no guarantee that this will always be the case.
4. *Heap*. This area is monitored and controlled by the system. Here is where we gain access via the dynamic memory allocation mechanism we saw earlier. Although in

modern systems this is a very big area responsible bookkeeping from our program is always required. We must always release any memory we no longer need so that other parts of our program will run smoothly when requesting memory.

5. *Stack*. This area is used for the local variables and the arguments of the functions in our program. Every time a function is called, a new stack 'frame' is allocated to store the locals and the arguments. This ensures that each invocation has its own copy of the variables allowing for recursive function calls (functions can call themselves). This stack 'frame' is released when the function exits, signaling the end of life for local variables. This is the reason we cannot return a pointer to a local variable.
6. *Command line arguments and environment variables*. This segment is at the top of the program's address space. The operating system stores these information for our program to use if needed. They are passed to our program as argument to the **main** function like this:

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

The first argument is the number of command line arguments passed to the program. This is always bigger than 0 because the first argument is the name of our program holding the index 0 in the string array that follows and contains all the arguments. In the example below

```
>our_program arg1 arg2
```

we have three arguments, and *argc* will be equal to 3:

```
argv[0] = our_program
argv[1] = arg1
argv[2] = arg2
```

The count of environment variables is not passed to our program. That is not a problem though. We can iterate through them until we find a NULL pointer like this

```
while (*env) {   // checking for NULL pointer
    std::cout << *env << "\n"; // print current variable
    ++env;          // move to the next
}
```

Summary

- In this chapter we were introduced to *pointers* and *references*.
- Pointers can be used as references to other variables.
- Reference variables are aliases to normal variables.
- Pointers can be used to handle dynamically allocated memory

Chapter 6: User defined variable type – Structures and enums

There are times when we need new types of variables that help us describe the data we handle more expressively. In C++ we have structures and classes for this purpose. In this chapter we will learn about structures.

- What they are and why we need them
- How to declare them
- How to use them

Why structures

Structures were inherited by the programming language C that C++ came to improve in the beginning. The simplicity behind their implementation and use make them somehow easier to grasp along with some basic ideas behind Object Oriented Programming.

Structures primarily are collections of attributes stored in variables, grouped together to form a new entity. Almost everything around us is a structure. We can hardly find a simple object; they are all made up of simpler ones. A person's name consists of a given name and a family name.

For the objects in an application, we need to know their position among other things. If we are talking about a 3D application, the position is stored in three variables.

In Chapter 3 we solved a similar problem by storing points in arrays of double precision numbers. That solution is not bad, but it leads to a cumbersome memory access as we saw in Chapter 5. Should we store this information in a one-dimensional array or in a two dimensional one with all the advantages of each implementation? And if we need more attributes for each point? Moreover if those attributes cannot be described by a double number?

To solve this problem, we will use *structures*. Structures let us group together variables of different types and treat them as a new entity with the specific attributes.

Structure declaration

By defining a structure, we are defining a new *type* of variable. Unlike simple variables that have a value, this type has attributes. Instead of one value this new type has many.

Here is how we declare a structure

```
struct object
{
    var_type v1;
    var_type v2;
    ...
};
```

The declaration of structures is usually done in header files, to be accessible from anywhere in our code. After the declaration of the structure, we can start using this new type.

Here is a classic example used in numerous application; a point in 3D space defined by its three coordinates:

```
// a point in 3D has three coordinates
struct point
{
    double x;
    double y;
    double z;
};
```

Structures are complete types and can be used as member variable types in other structures. Here is a specialized example with a game object:

```
// an imaginary object in a game
// using another struct as a member variable
struct space_ship
{
    // its location in space
    point location;
    // and its state (whatever that might be)
    int state;
};
```

The attributes of the structure are also called ‘members’ or ‘member variables. To access them we use the *member access operator* (.). We place a dot (.) between the structure variable name and the member variable name:

```
o.x = 10;
double x = o.x;
```

Arrays of structures

Now that we know how to create structures we will revisit the array of points we created before. We no longer need to step three numbers per point when indexing the array.

Declaring an array of structures is like declaring any other array

```
struct_type var_name[array_size];
```

Every array element is now a structure of the type we used in the declaration, whose members we can access using the *member access operator*. Here is our point sample using

```
void show_array_of_strutcs()
{
    // an array of points
    point the_points[10];

    // iterate through the points, and access their attributes
    for (auto i = 0; i < 10; ++i)
        the_points[i].x = 10;
}
```

structures.

```
// dynamic memory allocation with pointers
point* allocate_points(int num_points)
{
    // we simply request memory enough for our points
    // the call is similar to the one-dimensional array
    // it IS a one-dimensional array of points
    point* coords = new point[num_points];

    // and return the allocated memory
    return coords;
}

// accessing and modifying structured data
void do_stuff_with_points(point* points, int num_points)
{
    for (auto i = 0; i < num_points; ++i)
    {
        // we always move to the next point regardless of their size
        // no more complex memory access, simple and clear code
        double x = points[i].x; // fetch x
        double y = points[i].y; // fetch y
        double z = points[i].z; // fetch z
        // meaningless calculations but still quite representative
        // of how to access and modify our data
        points[i].x = x + y;
        points[i].y = y + z;
        points[i].z = z + x;
    }
}

// this is an abstract use of structures
void an_application_of_points()
{
#define NUM_POINTS 100
    // allocate NUM_POINTS (100) points
```

```
    point* the_points = allocate_points(NUM_POINTS);

    // perform some imaginary calculations
    do_stuff_with_points(the_points, NUM_POINTS);

    // always release the memory when we no longer need it
    delete the_points;
}
```

Now we can revisit the dynamic allocation of points we saw in the previous chapter.

We are not using any of the methods we did before. No need for three number steps or two-dimensional arrays. Clear code anyone can understand and maintain.

Assigning one structure to another

In C++ we can assign one structure variable to another. This way the member variables of one are copied to their respective counterparts on the other. This is fine for simple numeric values and arrays statically allocated, we get what we expect.

```
// this is how copying a struct works
void copy_simple_structs()
{
    point pt1;
    pt1.x = pt1.y = pt1.z = 1;

    // copy one structure to another
    point pt2 = pt1;
    // this will print "1,1,1"
    std::cout << pt2.x << "," << pt2.y << "," << pt2.z << "\n";
}
```

Things are a little tricky when we use member variables to store dynamically allocated arrays. Look at this structure definition:

```
// this is not as simple as a point!
struct complex_geometric_object {
    // dynamic array of points defining the object
    point* the_points;
    // number of points
    int num_points;
};
```

As we said copying one struct to another copies the member variables one by one. Let us create a function that allocates some memory and then copies the structure as we did in the simple example above:

```
void perform_shallow_copy()
{
#define NUM_VERTICES 5
    // this is a shape with 5 vertices (could have as many as we like!)
    complex_geometric_object pent;
    // allocate the points of the object
    pent.the_points = allocate_points(NUM_VERTICES);
    pent.num_points = NUM_VERTICES;

    // create a copy of our original geometric object
    complex_geometric_object pent_copy = pent;

    // now is time to release the dynamically allocated memory
    delete pent.the_points;
    delete pent_copy.the_points;
}
```

This results in two pointers pointing to the same memory location. As we said when we were talking about memory management, we must release the allocated memory when we no longer need it. In this case if we try to release both pointers the program will crash on the second release because re are releasing memory already released. In simple situations like this it is easy to avoid it by deleting one line of code. In real life though where do we end up with many copies passed from one function to the other who is responsible for releasing the memory? This is an accident (call me crash) waiting to happen.

This means that in cases like this we cannot rely on the automatic copy, which is called *shallow copy*, but we have to go one step further and allocate memory and copy the data ourselves or perform a *deep copy* as we call it. Here is a potential implementation of this:

```
// here is what we could do
// create a function that performs a complete copy
complex_geometric_object custom_copy(complex_geometric_object& original)
{
    // create a shallow copy of the original
    complex_geometric_object copy = original;
    // now we go one step further
    // allocate new memory for the points of the object
    copy.the_points = allocate_points(copy.num_points);
    // use a standard library function to copy the contents
    // of the original object to the copy
    memcpy(copy.the_points, original.the_points, copy.num_points *
sizeof(point));

    return copy;
}
```

```
void copy_complex_struct()
{
#define NUM_VERTICES 5
    // this is a shape with 5 vertices (could have as many as we like!)
    complex_geometric_object pent;
    // allocate the points of the object
    pent.the_points = allocate_points(NUM_VERTICES);
    pent.num_points = NUM_VERTICES;

    // use our custom copy function
    // shallow copy that occurs now is OK
    // we are copying a pointer that would otherwise be lost!
    complex_geometric_object pent_copy = custom_copy(pent);

    // now is time to release the dynamically allocated memory
    delete pent.the_points;
    delete pent_copy.the_points;
}
```

Creating a custom copy function lets us take care of in-depth copying of data so that we end up with two distinct structures that contain their own data, and no error occurs when we delete them. The shallow copy that is performed during the copy operation is fine. In this case we are copying and storing a pointer that would otherwise be lost and lead to memory leak.

Another consequence of the *shallow copy* is that when we modify the memory in one object it affects the other as well since they actually share it.

This behavior is normal and compatible with the language specification. The compiler has no way of knowing what to do for pointers that are dynamically allocated and initialized as opposed to static arrays that have fixed sizes and it copies whatever they contain at the time of the copy.

Pointers and references to structures

Now we are going to see how we can access the structures through pointers and references.

As with simple variables getting a pointer to a structure variable all we need is the *address off (&)* operator. Then to access a member we have two options. The most common is the *arrow* operator (->). This is the equivalent to the *member access* operator when we have a

pointer to a structure. The second and least common way is to dereference the pointer using the *asterisk* (*) operator and then use the *member of* (.) operator.

```
void pointer_to_struct()
{
    // declare a variable of type point
    point pt;
    // declare a pointer to a point
    // and assign the address of pt
    point* ppt = &pt;

    // set a member of pt
    pt.x = 10;
    // and read it through the pointer
    std::cout << ppt->x << "\n";
    // or
    std::cout << (*ppt).x << "\n";
}
```

Although the two are equivalent the second option is hardly ever used.

As we have seen C++ is not limited only to pointers to refer to an entity. It has *references*. References can be used for any entity in our program and *structures* are no exception. The advantage we have with references is we do not have to dereference the variable to access the members like we do with pointers. The syntax is identical to that of structure variable. There are few things we can say about the *references to structures* that we have not said already when we talked about references before.

```
// a point variable
point pt;
// and a reference to it
point& rpt = pt;

// we set the variable
pt.x = 123;
// and read the reference
std::cout << rpt.x << "\n";
```

These definitions lead us to the next section.

Structures and functions

It is evident that structures can be used as arguments to functions. We can pass them using any of the methods we have seen for built in types of variables: by value or by reference which can be done either using pointers or references.

Passing structures by value is something I do not really recommend. Structures can be quite big and complex. So, they will take up a lot of stack space and time to copy. Another

hint is using references for single structures and pointers for arrays. Consistency makes it easy to understand what a function expects just by looking at its declaration.

```
void move_to(point pt);
void move_to(point& pt);
void move_to(point* pt);
```

Accessing the member variables of the structure within the function is done as we saw before.

Enumerations

Enumerations (*enum*) is another way to create custom variable types in C++. They are named collections of named integral constants. Any variable declared to be of an *enum* type is limited to the values the specific type can have. Attempting to set it to a different value raises a compilation error.

Here is how we declare an *enum*

```
enum enum_name { value_name1, value_name2, ..., value_nameN };
```

By default, the first *value_name* is assigned the value '0', the second is assigned '1' and so on. We can change this very easily by assigning specific values to the named components.

```
enum enum_name{name1=value1, name2=value2, ..., nameN=valueN};
```

After we declare the *enum*, it is a valid type, and we can define variables based on it.

```
enum_name var_name = value_name;
```

Here is an example of how we might store the player's state in a game.

```
enum states { idle, walking, running, jumping };
states my_state = idle;
switch (my_state)
{
case idle:
    // we do some stuff
    break;
case walking:
    // we do some stuff
    break;
case running:
    // we do some stuff
    break;
case jumping:
    // we do some stuff
```

```
    break;
default:  // if we add more states in the enum
          // and forget to update the code
          // we get a runtime warning
    std::cout << "unhandled case!!!\n";
    abort();  // crash the program to force a fix
    break;
}
```

Enumerations are very useful because they limit the possible values a variable can take, and they let us use mnemonic names instead of numbers.

Summary

- Data structures
- Pointers to structures
- enumerations

Chapter 7: Classes

The idea of the *class* is almost the same as that of the *struct*. It is also used to define custom data types. Apart from giving us the ability to define our own data types it also introduced some other concepts that facilitate *Object Oriented Programming* and help us write more robust code.

The *class* is the foundation of object-oriented programming in C++. A *class* stores the template upon which the objects are created. In this chapter we will cover how we create classes, class inheritance, data encapsulation, and polymorphism.

Classes vs objects

People are often confused when talking about *classes* and *objects*. The distinction between the two is quite clear. It is like talking about humans and a specific person. Human is the class. It contains all the necessary information about a human being, general anatomy features for example, while the *object* or *instance* has values for the attributes defined in the class. Knowing the definition of a class we know how to *communicate* with an instance of that class. We can talk with a person, which is something we cannot do with instances of the class *lion*.

Getting started with classes

Classes and structures are almost the same in C++. Their internal structure is identical. Their only difference is the default access level. We will cover this issue later in this chapter. Here is how we define a *class*.

Defining a *class* is like defining a *struct*. The only differences are the *class* keyword, the *public:* access specifier and the two functions that have no return type.

```
class class_name {       // class name
public:                  // access specifier
    var_type var_name;   // member variable

    class_name();        // constructor
    ~class_name();       // destructor
    ret_type func_name();   // member function
};
```

The access specifier

The access specifier is a compiler only feature. It tells the compiler which members of the class are accessible from the code using it and which are for internal use only. By default, C++ defines all the members of a class as *private* unless we specify them as *public*. There is a third level of accessibility restriction called *protected*. This makes a lot of difference in *class inheritance*. For the rest of our code, it behaves like *private*.

Constructors and Destructors

The first thing we notice about a class is that it has functions. These are called *member functions* and they define the behavior of the class. The first function is the *constructor*. This function is invoked automatically by the compiler whenever an instance of the class is created hence the name *constructor*. The other function that starts with the tilde(~) is called *destructor* and is invoked when the object goes out of scope or is explicitly deleted. Both these functions have the same name as the class. Here is an example.

```
// a point in 3D class
class point
{
    double x;
    double y;
    double z;

public:
    point() {
        x = y = z = 0;
        std::cout << "point created\n";
    }
    ~point() {
        std::cout << "~point deleted\n";
    }
};

void main()
{
    // create some points
    point start_point;
    point end_point;
}
```

Here we see how the *default constructor* and the *destructor* are invoked. As we can see the constructor is a good point to initialize the state of the object and the destructor a good place for cleanup.

Constructor types

In the above example we see the *default constructor*. This constructor takes no arguments and in the example performs some standard initialization. We can also declare constructors that take arguments. The above example can have a constructor that takes three numbers, one for x, y, and z.

```
point(double _x, double _y, double _z) :x(_x), y(_y), z(_z) {
}
// and we can declare the variable like this
point start_point(1,1,1);
```

This constructor is also showing another method of initializing the variables. We are actually using their respective constructors. C++ supports this for built in types as well. We can have default values for the arguments like this

```
point(double _x=0, double _y=0, double _z=0) :x(_x), y(_y), z(_z) {
}
```

This constructor replaces the default constructor since it works with arguments as well as without them.

Another very important constructor is the *copy constructor*. This constructor takes one argument, a *constant reference* to an object of the same type. *Constant* means we will not modify its contents, so it can be a temporary object as well that will be passed by *reference*. This constructor is invoked when we initialize an object from the contents of another.

```
point(const point& pt) :x(pt.x), y(pt.y), z(pt.z) {
}

point p1(1, 2, 3);  // create the first point
point p2(p1);       // create the second using the first
point p3 = p2;      // this is equivalent to the above
```

The *copy constructor* can be found in either of the two forms presented here. The second form, although it looks like an assignment is actually a copy constructor. The compiler generates the same code in both cases.

Let us go back a little bit and remember the problem we faced in the previous chapter when we had two structures containing pointers thar pointed to the same memory. We saw that simply copying the value of the pointer was actually bad and we had to perform a custom copy operation. *Copy constructors* are a great point where we can correctly create an object based on an existing one. We can hide the deep copy in the copy constructor, leaving our code clean.

Destructors are automatically invoked when an object goes out of scope, or it is destroyed by calling the *delete* operator on a pointer. This is done behind the scenes by the compiler, and it is a good chance for a programmer to do some bookkeeping (actually memory management) work. If you recall from the previous chapter we had to manually delete the memory allocated and maintained in our structs. The introduction of destructors in C++ lets us put that code within a function that runs without any intervention so that our code is always clean, and our program works in harmony with the rest of the system.

```
// copy constructors are essential
// ...and destructors too
class memory_hungry {
    int* my_storage;
    int size;
public:
    memory_hungry() :my_storage(NULL), size(0) {
    }
    // without the copy constructor the pointer is copied
    // and we end up with two objects pointing to the same memory
    // copy constructors allow for seamless copy of the data
    // to a new location handled by the new class instance

    memory_hungry(const memory_hungry& other){
        allocate(other.size);
    }
    // destructor ensures we return the memory back
    // to the system when we no longer need it
    ~memory_hungry() {
        if (my_storage)
            delete my_storage;
    }
    void allocate(int s) {
        size = s;
        my_storage = new int[size];
    }
};
```

Member functions

C++ introduced the concept of *member functions*. These functions are declared as members of the class like the *member variables*. They are used to manipulate the variables and they have access to the *private* variables as well.

The *member functions* define the overall behavior of the class and the objects based on it.

Keeping together the data and the behavior is what we call *encapsulation*.

Here is an example of a member function in the class point we introduced before. It moves (translates) a point by a given extent in each axis.

```
// member function that moves (translates) the point
void translate(double _x, double _y, double _z) {
    x += _x;
    y += _y;
    z += _z;
}
```

Encapsulation led to the next concept '*data abstraction*'. This means that a class exposes only the interfaces required and keeps the implementation details hidden from the user.

All these are achieved using the *private, protected,* and *public* specifiers. All the member functions in our class have unrestricted access to all class member private or public.

Here is how we can modify our *point* sample class:

```
// a point in 3D has three coordinates
class point
{
private:
    // hide implementation details
    double coordinates[3]; // use an array instead of distinct variables

public:
    point(double _x=0, double _y=0, double _z=0) :coordinates{_x,_y,_z} {
    }
    point(const point& pt):coordinates{ pt.coordinates[0], pt.coordinates[1],
pt.coordinates[2] } {
    }
    ~point() {
    }
    // we use functions to access the data
    // we still support x,y and z that everyone understands
    // and we use references to directly set coordinates
    double& x() {
        return coordinates[0];
    }
    double& y() {
        return coordinates[1];
    }
    double& z() {
        return coordinates[2];
    }
    // member function that moves (translates) the point
    void translate(double x, double y, double z) {
        coordinates[0] += x;
        coordinates[1] += y;
        coordinates[2] += z;
    }
};

void some_ function()
{
    point p1(1, 2, 3);  // create a point
```

```
    p1.y() = 10;         // setting a coordinate
    std::cout << p1.y() << "\n";   // and reading it
}
```

This way the rest of our code has only access to the interface of the class. We can modify the implementation of our class and our program will not need any change. The rule of thumb is to keep in the public interface only the functions or data that are absolutely necessary and move everything else in private.

Function overloading

We can overload member functions as we can with any other function. The rules of overloading are the same. As long as the functions have different argument lists, they can coexist in the code.

Constructors are the most characteristic cases of function overloading in classes. Apart from the *default* constructor we have seen the *copy* constructor and the constructor that takes arguments to initialize member variables.

The use of *const*

We saw *const* when we talked about variables in chapter 1. The *const* keyword can be used in other places as well giving special meaning to our code, while letting the compiler optimize our code and keeping us protected from bad programming habits.

Generally speaking, the keyword *const* tells the compiler that something is constant or, it safe to consider it as a constant.

Constant function arguments

When we declare function arguments as *const*, the compiler knows that we will not modify their value inside the function. This is particularly handy when we pass references or pointers to variables. If we make a mistake and try to modify their values it raises a compilation error, and we catch it in the very early stages of the development.

```
// this function takes a reference to a constant string
void print_out_string(const std::string& str)
{
    // this is a legitimate use of the constant variable
    // reading its value is fine
    std::cout << str << "\n";

    // if we try to modify the string
    // a compilation error is generated
    // uncomment this line and try to compile
    // str = "some other value";
}
```

Another benefit is that we can use a temporary object when calling this function:

```
void call_print()
{
    std::string str("hello world");
    // we can call the function passing it some variable
    print_out_string(str);
    // or we can create a temporary argument on the stack
    print_out_string("temporary value");
}
```

Functions returning const

These functions return variables that cannot be modified. Usually they return references or pointers to variables we can only read. That is very handy when we want to give read only permission to the member variables of our classes.

```
class solid_body {
    double m_mass;
public:
    solid_body(double m = 0) :m_mass(m) {}

    // this function gives read only access to the object's mass
    const double& mass() { return m_mass; }
};
```

Member functions declared as const

C++ lets us declare functions as being *const*. When a function is declared as *const* it means that it cannot modify any of the member variables of the class, nor can it call any other function that might do so. These functions are required when we need to perform some calculations on objects that are constant like the string we saw earlier.

The std::string class returns the length of the string with the function *length()*. This function does not modify the contents of the string and is declared as *const*. So we can call it is our string is constant.

Here is another example:

```
class point2d {
    double x;
    double y;
public:
    point2d(double _x = 0, double _y = 0) :x(_x), y(_y) {}
    // the following function reads member variables
    // and should no modify them
    double get_x() const {
        std::cout << "constant function:" << x << "\n";
        return x;
    }
    // the following function modifies member variables
    double get_x() {
        x = 10;
        std::cout << "non constant function:" << x << "\n";
        return x;
    }
};

void demo_const_function()
{
    point2d p(0, 0);
    const point2d cp = p;
    // notice which overloaded function is called
    // based on whether the object is const or not
    std::cout << p.get_x() << ", " << cp.get_x() << "\n";
}
```

Operator overloading

Classes define new data types. It is only logical that we must be able to define how these objects interact with each other .

Standard operators are functions built in the language. They are designed to take as arguments the standard data types. If we try to add two custom objects the compiler will object. There is no internal function that can handle this operation.

Since operators are functions, we can *overload* them just like any other function and write our own function that accepts the arguments we want. The compiler will then use our function and perform the operation.

Here is the language definition and an example

```
class_name operator op(const class_name& second_operand){
    return class_name (/* handle accordingly */);
}

// adding two points creates a new point by adding coordinates
point operator+(const point& pt){
    return point(coordinates[0] + pt.coordinates[0],
        coordinates[1] + pt.coordinates[1],
        coordinates[2] + pt.coordinates[2]);
}

// so, this code is perfectly valid
point p1(1, 2, 3);  // create a point
point p2(2, 3, 4);  // create another point
point p3 = p2+p1;   // and now add them to create a third
```

If this feature is not available in a programming language we have to create a function that takes two objects and returns their sum. This is the way it was done in C.

Dynamic allocation

Everything we have seen so far about memory allocation and dynamic memory management applies to classes as well. We can allocate dynamically allocate and delete objects according to our needs using the *new* and *delete* operators.

This is a good opportunity to revisit object construction and destruction and clarify the difference between a class and an object.

New

With operator new we create new objects of a specific class. The whole process can be represented with the process of creating a car. The class is the blueprint with the details of how to create a car. The car creates is the object or instance of the class. The constructor may be considered to be the order which is invoked with the options the customer has selected, for instance the color of the car.

```
point* ptr = new point(1, 2, 3);
point* ptr2 = new point(*ptr);
```

This is similar to the creation of a point we saw earlier. When a new object is created we actually invoke the constructor. So, we can pass it initialization information. Notice the dereference operator in the second call. We dereference the pointer and the compiler will generate a reference to the object. We can also create a constructor that initializes the object from another object's pointer.

We can allocate more objects of course. Creating an array of objects will invoke the constructor for each one of them.

```
// simple array allocation
point* dp = new point[5];
// allocation with initializers
point* dp = new point[5]{ {1, 4, 5},{2, 5, 6},{3, 6, 7},{4, 7, 8},{5, 8, 9} };
```

Although we can use initializer lists for object construction it is not very convenient. So, a default constructor or one with default initialization parameters is really handy.

Delete

The delete operator must be used when we no longer need the objects we created with new. They take up memory that we must return to the system when we no longer need it.

Calling the delete operator we invoke the destructor. In the case of an array of objects the destructor is called for each one. Here is how we delete one object or an array of objects:

```
// delete an array of objects
delete [] dp;
// delete one object
delete ptr;
```

Summary

- Classes and objects
- Constructors and destructors
- Member functions
- Operator overloading

Chapter 8: Inheritance

Inheritance is a core concept in object-oriented programming. It allows us to create new classes by reusing and expanding existing ones.

In this chapter we will cover:

- What is inheritance
- Virtual functions, abstract classes, and interfaces
- Multiple and multilevel inheritance
- Types of inheritance

Inheritance

A cube is a solid object. A sphere is also a solid object. We see there is a generalization/specialization hierarchy. They are both solid objects, which means they share some characteristics, yet they defer is so many ways we need to create two different classes for them. We also need a way to keep these classes closely related because of the things they have in common. Either object is a valid representative of a solid.

We can represent this hierarchy with class inheritance. *Inheritance* is the mechanism within C++ the enables one class to derive properties and behaviors, or *inherit* them, from another class. It is one of the most important features in OOP.

Base class or *parent class* is the class that we derive characteristics from

Sub class or *child class* is the class that inherits the characteristics

This is the syntax we use to create a subclass:

```
class subclass_name : access_specifier base_class_name
{
    // class declaration
};
```

The *access_specifier* limits what we can access in the base class. This is something we will see later in this chapter. For the time being we will keep everything *public* and fully accessible to simplify the samples and understand how inheritance works.

A cube and sphere have some common attributes. We will use mass, and position for now. These attributes are common among all solid objects. So, we are going to create a *solid_object* class that will hold the common attributes and then two special classes to differentiate between cube and sphere.

```
// some fundamental stuff in 3D space
class point {
protected:  // only child classes can access these
    double x, y, z;
public:
    point(double _x = 0, double _y = 0, double _z = 0) :x(_x), y(_y), z(_z) {}
    ~point() {}
};

// a solid object definition
class solid_object {
protected:
    double mass;
    point position;
public:

    solid_object(double m = 0) : mass(m) {}
    virtual ~solid_object() {}
    // pure virtual function
    virtual double volume() = 0;
};

// the cube is a special solid object
class cube : public solid_object {
protected:
    double edge_length;
public:
    cube(double m, double el) :solid_object(m), edge_length(el) {}
    virtual ~cube() {}
    virtual double volume() { return edge_length * edge_length * edge_length;
}
};

// so is a sphere
const double pi = 3.1415926;
class sphere :public solid_object {
protected:
    double radius;
public:
    sphere(double m, double r) :solid_object(m), radius(r) {}
    virtual ~sphere() {}
    virtual double volume() { return 4 * pi * radius * radius * radius / 3; }
};
```

The basic concept behind *inheritance* is the ‘is-a’ relationship between base and derived classes. Both the cube and the sphere *are* solid objects. This means we can store both cubes and spheres in an array of solid object pointers. We cannot store objects because they might require different space to store, but since pointers have standard size and both object types are solid objects we can mix them.

```
solid_object* objects[2];
objects[0] = new cube(1, 2);
objects[1] = new sphere(1, 3);
```

Here we saw the *virtual* keyword. So let us see what it is for.

Virtual functions

Virtual functions are the key to objects behaving according to their specific type regardless of the way we call them. In the previous example we know that the volume of the sphere is calculated differently than the cube. Yet it is good to have a function called *volume* in both classes that does the calculation. This is the actual concept of *polymorphism*.

```
double volume1 = objects[0]->volume();
double volume2 = objects[1]->volume();
delete objects[0];
```

The virtual function mechanism ensures that the correct *volume* function is called although the 'objects' are 'solid_object' pointers. This means that a pointer to an object behaves like the type it actually is despite the fact that it was accessed via a pointer of a class higher in the hierarchy.

The class destructor is usually the most critical function in polymorphic classes. Most of the times we use base class pointers to store our objects and it is important to call the right destructor when we want to release them.

Abstract classes

The *solid_object* class in our example is very convenient when we talk about these objects, but we cannot create solid objects in general. We can only create *cubes* or *spheres* or any other type we need to define in our program. Generalizations are *abstract interfaces* to real world objects.

In C++ we can have classes that cannot be instantiated but only be used as base classes that provide interfaces for the derived ones. The user cannot create objects of that type. They serve only as interfaces.

An *abstract class* in C++ is a class that has at least one *pure virtual* function. A pure virtual function is specified by setting it equal to 0 in its declaration like this:

```
virtual double volume() = 0;
```

We can change the *solid_object* base class to have this *pure virtual* function. Now none can instantiate an object of this class directly as a variable on the stack or by calling *new*. We can only have a reference or a pointer to this type initializing it from a *cube* or a *sphere*.

The *solid_object* now has become an *abstract class* and can be used as an *interface*. The classes that can be instantiated are called *concrete classes*.

Abstract classes provide a unified interface for all external access to the objects. The classes are designed similarly, and the objects behave in a uniform manner. Adding more objects requires minimum effort and everything works seamlessly.

Run Time Type Information (RTTI)

Run Time Type Information is what we can use to determine the exact type of an object if we are given a pointer of a more abstract type. Through RTTI C++ provides us with enough information about the actual type of the object in question.

The most common way of getting a pointer to an object of the actual type is using the *dynamic_cast* function. The syntax of this function looks a little weird at the moment. That is because it is a *template* function. We will talk about *templates* in the next chapter and then these things will become clear. For now, we are just using them.

```
class_name* s = dynamic_cast<class_name *> (object_pointer);
if (s == NULL) ...
```

If *dynamic_cast* fails it returns a NULL pointer so we can tell what the pointer is not. This trial and error until we get to the right type makes RTTI inconvenient and so it should be used only if anything else fails.

Another limitation of RTTI is that the classes must be *polymorphic*. That means they must have at least one *virtual* function. This case is not so common because polymorphism and inheritance go side by side, but still, it is possible since the language supports it.

Access control

Here we are going to analyze *access control* in C++. We saw the *public, protected* and *private* keywords earlier, but we only scratched the surface of their usage. Now that we have seen inheritance we can go in more depth with access control.

What the rest of the world sees

With access specifiers we can limit what the rest of our code can access within our class. The description here applies to all our code except form the derived classes. Inheritance has its own set of rules, and we will examine them later.

As we saw in the previous chapter, everything declared *public* in our class is accessible by the rest of the code, while everything declared as *protected* or *private* is not accessible.

Two very simple rules and we are done. Things are little more complicated with inheritance. Derived classes add one more level of access control which modifies base class access levels. They are not affected by the access specifier themselves. They always have access to members of the base class controlled by the ace specifiers in the base. Inheritance access specifiers have meaning for the rest of the code.

Public inheritance

This occurs when the derived class in declared with the *public* keyword from the base class as in this example

```
class sphere :public solid_object
```

In *public* inheritance all the members of the base class retain their access level as members of the derived class.

Protected inheritance

Here we declare the inheritance as *protected*.

```
class sphere :protected solid_object
```

In this type of *public* and *protected* members of the base class become protected members of the derived class.

Private inheritance

For this we use the *private* keyword.

```
class sphere :private solid_object
```

All members of the base class are now *private* in the derived class.

Public inheritance needs no explanation. We clearly create an is-a relationship between the classes, and we go from a general description to a more specialized one.

Private inheritance on the other hand allows us to inherit everything the base class has to offer without letting the rest of the world know about it. This enables us to have access to the base class protected members without exposing them to the public.

Protected is a less restrictive inheritance method. It allows access to the base class for the descendants of the derived class but hides it from the rest of the world.

Private and *protected* inheritance are closer to what is called *composition*. In composition we have a member variable of the type we derive from, instead of deriving or class from that. Most of the times this may be sufficient but limits us from accessing protected members of that class. Another benefit we have from inheritance is memory efficiency since composition takes more memory. In any case we must count the pros and the cons and design our classes accordingly.

```
class base {
public:
    base() {}
    virtual ~base() {}
    int x;    // a public variable
    void do_job() {   // and a public function
        x = 0;
        y = 1;
        z = 2;
    }
protected:
    int y;  // a protected variable
private:
    int z;  // and a private variable
};

class public_derived : public base {  // public inheritance
public:
    public_derived() {}
    virtual ~public_derived() {}
    void do_sth() {
        do_job();  // public and protected members are accessible
        x = 0;
        y = 1;
        //z = 2;  private is not accessible
    }
};
class protected_derived : protected base {  // protected inheritance
public:
    protected_derived() {}
    virtual ~protected_derived() {}
    void do_sth_else() {
        x = 0;     // again public and protected are accessible
        y = 1;
        //z = 2;   while private is not
    }
};

// deriving from a protected derived class
class after_protected : public protected_derived {
public:
    after_protected(){}
```

```
    virtual ~after_protected(){}

    void after_thought() {
        x = 100; // we still have access to the base class
    }
};

class private_derived : private base { // private inheritance
public:
    private_derived() {}
    virtual ~private_derived() {}
    void do_sth_new() {
        x = 0;
        y = 1;
        //z = 2;
    }
};

class after_private : public private_derived {
public:
    after_private() {}
    ~after_private() {}
    void after_all() {
        // x = 1;    we no longer have access to the base class
    }
};

int main()
{
    public_derived object1;
    protected_derived object2;
    private_derived object3;

    object1.do_job();    // base members are accessible
    // object2.do_job(); // base members are not accessible

    base* optr1 = &object1;  // object1 is base
    // base* optr2 = &object2;  // object2 has no accessible interface to base

    return 0;
}
```

Multilevel and multiple inheritance

We create *multilevel* inheritance when we need to add more specialization to our classes. We can start with a class *vehicle* which we specialize with a class *car* which is in turn

specialized with classes like *racing_car, sports_car, family_car* and so on.

```
// multilevel inheritance
class vehicle {
public:
    vehicle() { std::cout << "vehicle\n"; }
    virtual ~vehicle() {}
    virtual void print() = 0;
};
// car is a vehicle
class car :public vehicle {
public:
    car() { std::cout << "car\n"; }
    virtual ~car() {}
    virtual void print() {
        std::cout << "i am a car\n";
    }
};
// sports_car is a car, and a vehicle
class sports_car :public car {
public:
    sports_car() { std::cout << "sports car\n"; }
    virtual ~sports_car() {}
    virtual void print() {
        std::cout << "i am a sports car\n";
    }
};
```

Multiple inheritance occurs when a class is derived from more than one classes. Multiple inheritance adds attributes to our classes. Instead of deriving from a class we can use composition as we saw earlier, but that prohibits us from accessing protected members of the class and may add complexity in our code. Careful design is required to turn these language features into useful tools.

```
class made_by_me {
public:
    made_by_me() {}
    virtual ~made_by_me() {}
    void print() {
        std::cout << "my product\n";
    }
};
class my_car :public car, public made_by_me {
public:
    my_car() {}
    virtual ~my_car() {}
};
```

```
int main() {
    my_car c;
    // access the car derived function
    c.car::print();

    return 0;
}
```

Virtual inheritance

Multiple inheritance may lead to odd situations where multiple instances of a class appear in the inheritance hierarchy. Consider the case:

```
class a_base {
public:
    a_base() { std::cout << "a_base\n"; }
    virtual ~a_base() { std::cout << "~a_base\n"; }
    int x;
};

class b_base :  public a_base{
public:
    b_base() { std::cout << "b_base\n"; x = 10; }
    virtual ~b_base() { std::cout << "~b_base\n"; }
};

class c_base : public a_base {
public:
    c_base() { std::cout << "c_base\n"; x = 20; }
    virtual ~c_base() { std::cout << "~c_base\n"; }
};

class d_class : public b_base, public c_base {
public:
    d_class() { std::cout << "d_class\n"; }
    virtual ~d_class() { std::cout << "~d_class\n"; }
    void print() {
        std::cout << b_base::x << "," << c_base::x<<"\n";
    }
};
int main() {
    d_class dc;
    dc.print();
    return 0;
}
```

Both *b_base* and *c_base* are derived from *a_base*. Since *d_class* is derived from these two classes when we run the program we will see that the constructor of *a_base* is invoked twice. This shows that we have two instances of *a_base*. At the end *d_class* has two 'x's with possibly different values. *Virtual inheritance* we can resolve this issue. Here is how:

```
class b_base : virtual public a_base ...
class c_base : virtual public a_base ...
```

We declare the inheritance as *virtual*. This makes the two instances of *b_base* and *c_base* refer to the same instance of *a_base* at run time. This resolves the ambiguity, and we no longer need the odd-looking *b_base::x* or *c_base::x* to disambiguate between the two. Having one instance of *a_base* conserves memory as well.

Friends

A *friend class* or a *friend function* can access the private and protected members of a class.

```
class f_base {
private:
    int prx;
protected:
    int pry;
public:
    int puz;

    f_base() {}
    ~f_base() {}
    // declare friend function
    friend void my_friend_function();
    // declare friend class
    friend class my_friend_class;
};

void my_friend_function()
{
    f_base fb;
    // access private variable
    fb.prx = 100;
}

class my_friend_class {
private:
    f_base fb;
public:
    my_friend_class() {
        // access protected variable
        fb.pry = 200;
    }
};
```

The main problem with *friends* is that the classes and functions we want to make friends must be known in advance. So, this technique can only be implemented while we are designing our program.

The second thing we need to consider is that they break the whole idea of encapsulation in OOP. They better be applied as little as possible and only when we can find no workaround.

Protecting our implementation from malicious use

Now that we have cleared the meaning of access specifiers it is time to see how we can use them to improve the robustness of our code.

First we must keep the details of our classes hidden. This means that data members should be private. Any derived class or other code that needs to have some access to them should only do it through member functions that get or set their values. This protects us from errors, mostly unintended, but still capable of breaking our program and making it crash.

A class should only expose through the public interface only those functions that are necessary to communicate with the rest of the program. Take a look at this example:

```
// a vector in 3D space
class vector3DA {
    // the coordinates of the vector are private
private:
    // here we keep three distinct variables to hold the coordinates
    double x, y, z;
public:
    // the public interface of the class
    //
    // the basic constructor
    vector3DA(double _x = 0, double _y = 0, double _z = 0) :x(_x), y(_y), z(_z)
{}
    // the copy constructor
    vector3DA(const vector3DA& v) :x(v.x), y(v.y), z(v.z) {}
    // the destructor
    ~vector3DA() {}
    // move / translate the vector
    void move(const vector3DA& v) {
        x += v.x;
        y += v.y;
        z += v.z;
    }
    // get the coordinates of the point
    double get_x() { return x; }
    double get_y() { return y; }
    double get_z() { return z; }
};
// taking a different approach
class vector3DB {
    // the coordinates of the vector are private
private:
    // here though we have an array of doubles to hold the coordinates
    double coords[3];
```

```
public:
    // the public interface of the class
    // is the same as the class above
    // the basic constructor
    vector3DB(double _x=0, double _y=0, double _z=0) :coords{_x, _y, _z} {}
    // the copy constructor
    vector3DB(const vector3DB& v) :coords{ v.coords[0], v.coords[1],
v.coords[2] } {}
    // the destructor
    ~vector3DB() {}
    // move / translate the vector
    void move(const vector3DB& v) {
        coords[0] += v.coords[0];
        coords[1] += v.coords[1];
        coords[2] += v.coords[2];
    }
    // get the coordinates of the point
    double get_x() { return coords[0]; }
    double get_y() { return coords[1]; }
    double get_z() { return coords[2]; }
};
```

Here we have two implementations of a simple vector in 3D space. Keeping the data hidden from the outside code we can select the implementation that best suits our needs based on the mathematical infrastructure our program might use. We may even change the implementation and no code has to be modified in the program.

Almost the same as driving a car. All the cars have the same user interface, two or three pedals, a steering wheel, a gear selector and so on. We do not really care about the engine, or any other component hidden under the hood as long as we get the expected results from the user interface, accelerate, break, turn and so on.

Order of creation

Now that we have covered inheritance we must clarify how objects are created. Understanding how objects are created is crucial, especially when we have virtual functions. Misconceptions and lack of understanding of creation order leads to programming errors that are hard to find.

When we instantiate a derived object, the base class is instantiated first. If you run the *multiple inheritance* sample presented earlier you will see that the *vehicle* class messages are printed before those of the *car* class as we create a *car* object.

This means that when we invoke a virtual function from the constructor this will end up in the base class implementation of the function although we are creating a derived class. This is because the derived class is not initialized yet and so its virtual table is not accessible.

Summary

- What is inheritance
- How we can benefit from virtual functions
- Abstract classes and interfaces
- Run Time Type Information
- Access specifiers in inheritance
- Multilevel and multiple inheritance
- Virtual inheritance
- Friend classes and functions
- Why we need to hide the inner workings of our classes

Chapter 9: Templates

A *template* in general is a mold or a stencil we use to create copies of a shape. The same applies in C++. Templates are used by the compiler to generate code. This applies both to functions and classes. C++ has the tools we need to design and write our code using *generic types*. So, our program can work with many data types without rewriting or loading it with specific code just in case we might need it.

What is a template

Templates are one of the most powerful features of C++. They are the blueprints the compiler uses to generate code at compile time as instances of them, as objects are instances of classes at run time. A template takes one or more parameters that you must supply when you instantiate them.

Writing code for templates is called *generic programming*. This type of programming is independent of data types. We develop our code once and it applies for many data types.

Unlike traditional OOP that is based on polymorphism, template programming is *parametric polymorphism*. Instead of using specific datatypes the code is developed to handle values without depending on their type. The functions and the data types are called *generic functions* and *generic types* respectively.

Function templates

A function template is similar to a function. But with a key difference that makes the template a lot more versatile. Normal functions work with specific data types while template functions work with a set of data types.

When we need to perform the same operation with different data types we can overload a function and have two versions accepting different types. Perfectly valid solution and the compiler will sort out which function to call depending on the arguments. Only now we have two functions to maintain.

Developing generic functions with the use of templates overcomes the problem of multiple code.

We start with the keyword *template* followed by the template parameters enclosed in the `<>`. The function declaration that follows uses the template parameters as data types wherever required. Typically, a function template is declared like this:

```
template<class T>
T someFunction(T arg)
{
    T result = arg;
    ...
    return result;
}
```

When we call this function the compiler deduces the type by the argument we pass and generates the function code for this type. Here is a template function example:

```
template<typename T>
T absolute_value(T a)
{
    if (a >= T(0))
        return a;
    return -a;
}
int main()
{
    std::cout << absolute_value(-3.5) << "\n";
    std::cout << absolute_value(4) << "\n";
}
```

The first call will force the compiler to generate code for *double* and the second for *int*. in this example we used the keyword *typename* instead of *class* in the template arguments. It is the older keyword used for template arguments and it can still be used.

Our program has two overloaded functions, but we have only one function to maintain.

Class templates

The idea behind class templates is the same as for function templates. There are times when we need similar classes for different data types. A characteristic example is a 2D drawing application. It requires *points* with double precision to store design data, while for display purposes it requires points with integer coordinates. A point is conceptually the same and needs the same functionality. The only difference is coordinates data type depending on the application.

C++ allows us to define class templates like function templates. We start with template declaration and parameter list followed by the definition of the class using generic types.

```
template<class T>
class point {
    T data[2];
public:
    point(T x = T(0), T y = T(0)) : data{ x,y } {
    }
    void print() {
        std::cout << data[0] << ", " << data[1] << "\n";
    }
};

int main()
{
    // design data, double precision
    point<double> pt( 10, 21.3);
    pt.print();
    // screen coordinates, integers
    point<int> ipt(31, 23);
    ipt.print();

    return 0;
}
```

In the above example we instantiate classes (not objects) for *double* and *int*. Then we instantiate the respective objects (points). The compiler generates the two distinct classes before allocating memory for the points themselves.

Template specialization

The way we presented templates so far they seem to limit us in one version fits all. This is not the case though. We can have special version for a specific type if we need to differentiate the behavior of our code.

In the previous sections we developed a template function for calculating the *absolute* value of number, and a template class for storing *points*. Here we will 'develop' custom versions for the integer values.

```
template<>
int absolute_value<int>(int a)
{
    std::cout << "int specialization\n";
    if (a >= 0)
        return a;
    return -a;
}
```

We start with the *template* keyword and an empty parameter list. Then the function declaration states that this specialization is for the *int* type.

Similarly for the class specialization we have

```
template<>
class point<int> {
    int data[2];
public:
    point(int x = 0, int y = 0) : data{ x,y } {

    }
    void print() {
        std::cout << "integer point " << data[0] << ", " << data[1] << "\n";
    }
};
```

Template metaprogramming

The C++ compiler performs computations when no variables are involved. This simplifies the generated code and speeds up program execution. Take a look at the code:

```
const int i = 10;
int k = 2 * i;
int j = 7 + 8;
```

All these calculations will be evaluated at compilation time. The general idea is that the compiler evaluates as much as possible leaving only the variable parts for run time.

Templates can be used to force the compiler to perform more complex computations. The following function raises *base* to the power of *exp*. This code is executed in run time and does the job fine.

```
int power(int base, int exp)
{
    int r = 1;
    while (exp)
    {
        r *= base;
        --exp;
    }
    return r;
}
```

The following initialization though is static. It does not depend on any variable. With this code it will be done at run time resulting in a significant time penalty if it is executed frequently.

```
int cube_volume = power(4, 3);
```

For reasons that have to do with our program design we want to keep this call or something similar. The need to optimize execution speed means we need to have the compiler do the evaluation at compilation time.

This template will do the job.

```
// template to calculate powers of integers
// recursively defined struct
template<int base, int exp>
struct powerOf
{
    enum { value = base * powerOf<base, exp - 1>::value };
};
// special case when we reach 0
template<int base>
struct powerOf<base, 0>
{
    enum { value = 1 };
};
```

We can change the variable initialization with this:

```
int cube_volume = powerOf(4, 3);
```

Generating the code of the template and evaluating the constant calculations will force the compiler to generate the value ‘81’. The resulting code is clear and fast.

Template metaprogramming is a huge topic by itself and has many applications, especially when we need to initialize static data. In the past we used to spend a significant amount of execution time, especially during program startup to initialize variables that now we use the compiler to do so.

Summary

In this chapter we covered templates

- What they are
- How to develop function templates
- Class templates and their usage
- Specializing templates
- Template metaprogramming

Chapter 10: The C++ standard library

By design C++ provides only the basic mechanisms to convert our ideas into machine executable actions. There are no built-in functions that perform tasks as reading keyboard input or producing any form of output. All these tasks are completed with libraries.

In the beginning the library that came along with the language was very limited and C++ relied a lot on its compatibility with C. The nature of the language encouraged many developers to create libraries of classes that were covering many disciplines.

In 1993 Alexander Stepanov presented a new library, based on generic programming. This library was initially called Standard Template Library or STL for short and was adopted by the ISO standardization committee in 1994. Years later its name was changed to Standard Library.

The C++ Standard Library is a very big topic that cannot be covered in a chapter. Dedicated books with thousands of pages have been written about it. Our aim here is to cover the basics required to understand the development of a computer game and an application in general.

Introduction and general concepts

By the time if this writing the most common version of the standard is C++14. It is supported by almost all the common compilers. The next version C++17 is not widely supported, but it is supported by Microsoft Visual C++ in Visual Studio 2019.

Standardization is hard and slow process. Adapting to the changes in the standard is even slower.

The C++ standard library relays heavily on generic programming. it is implemented in *header* files, and you do not need to link your code with external libraries. All the identifiers of the library are members of the *std* namespace.

```
// the include header for the output functionality
#include <iostream>

int main()
{
    // accessing the object in the std namespace
    std::cout << "Hello World!\n";

    return 0;
}
```

Utility library

We are starting our exploration in the Standard Library with *Utilities*. These are small classes and functions that are fundamental elements of other classes and algorithms .

pair and tuple

A *pair* is used to store two values and treat them as a unit. It is used in many places inside the library, especially in containers that map a key to a value. Here is the definition

```
template <class _Ty1, class _Ty2>
struct pair { // store a pair of values
    using first_type  = _Ty1;
    using second_type = _Ty2;
...
```

It is implemented as struct to keep the members public.

```
std::pair<int, double> id(1, 2.5);
std::cout << id.first << ", " << id.second << "\n";
std::pair<int, double> id2 =id;
id2.first = 0;
id2.second = 100;
std::cout << id2.first << ", " << id2.second << "\n";
// the ordering is done using the first element
if (id > id2)
    std::cout << "id is bigger\n";
else
    std::cout << "id is smaller\n";
```

The *tuple* was created as an extension for *pair*. It can have a variable number of elements and, just like *pair*, they can be of any type. The definition of tuple is in *<tuple>* header which we must *#include* in our code. The definition of tuple relies on the variable number of arguments of templates with some help of recursion.

```
template <class _This, class... _Rest>
class tuple<_This, _Rest...> : private tuple<_Rest...> { // recursive tuple
definition
public:
    using _This_type = _This;
    using _Mybase    = tuple<_Rest...>;
...
```

The elements of the tuple are ordered we access them using their position:

```
std::tuple<int, int, float> t(1,2,float(3));
t = std::make_tuple(3, 4, float(5));
std::get<2>(t) = 100;
std::cout<<std::get<0>(t)<<", "<<std::get<1>(t)<<", "<<std::get<2>(t)<<"\n";
```

Smart pointers

We have seen that pointers are important in dynamic memory allocation. The problem is that we must release them but there are cases where we do not know when to release them. The language does not provide any mechanism to find unused memory to return to the system.

There are some workarounds in the library though. These are the *smart pointers*. We call them *smart* because they have some built-in *intelligence*, to help programmers overcome the problems normal pointers generate.

There are more than one type of smart pointers, depending on the application. Misusing them can still lead to problems and create errors.

The first class of smart pointers is *shared_ptr*, which stands for shared pointer. It can be used as normal pointer to an object. You can assign, copy or any other operation and access its members using * and ->. Take a look at this example:

```
#include <iostream>
#include <memory>

class worker {
public:
    int i;
    worker() {}
    ~worker() {
        std::cout << "worker destroyed\n";
    }
};

int main()
{
    // create a place to store the data
    std::shared_ptr<worker> w[3];
    // create a local scope
    {
        std::shared_ptr<worker> sw(new worker);
        sw->i = 100;  // accessing the member variable
        w[0] = sw;
        w[1] = sw;
        w[2] = sw;
        // the smart pointer is not deleted here, at the end of scope
        // comment out the above 3 lines and see the result
    }
    std::cout << w[2]->i << " ---------\n";
    std::cout << "---------\n";
    // the smart pointer is deleted after this point
    // when the references to it go out of scope

    return 0;
}
```

Shared pointers are declared in <memory>. We declared a shared pointer for our *worker* objects. First we declared an array to store some pointers. Then in a local scope we create a pointer and initialize it to a *worker* object pointer with *new*. The members of the object are accessed as we normally expect. Storing the allocated pointer in the array increases its internal counter, so when we exit the local scope the destructor is not invoked. The destructor is called after the *return* statement at the end of the function.

If we put a comment at the three lines that sore the pointer in the array, the destructor is called when the allocated pointer *'sw'* goes out of scope.

The shared pointer has given us the liberty to skip deleting the object at all and the robustness of deleting it when it was no longer needed.

Now take a look at this code:

```
class worker {
public:
    int i;
    std::shared_ptr<worker> leader;
    std::shared_ptr<worker> partner;

    worker(int _i=0) : i(_i){
        std::cout << i << " worker created\n";
    }
    ~worker() {
        std::cout << i << " worker destroyed\n";
    }
    void talk() {
        std::cout <<  i <<" i am a worker\n";
    }
};

std::shared_ptr<worker> create_team()
{
    std::shared_ptr<worker> l(new worker(1));
    std::shared_ptr<worker> p(new worker(2));
    l->partner = p;
    p->leader = l;

    return l;
}

void test_weak_pointer()
{
    std::shared_ptr<worker> lead = create_team();
    std::cout << "--------\n";
    // make him 'talk'
    lead->partner->talk();
}
```

Everything seems to work but there is a problem. Although we are using shared pointers the objects are not deleted at the end.

The two objects refer to each other as we created a working partnership. So, when the function exits, and everything goes out of scope the pointers are not deleted as we might expect. Shared pointers do not seem to apply when cross reference occurs. They create a strong bond between the objects that keeps their usage count greater than zero, so as they get out of scope they do not release the memory.

The solution to this problem is the use of a weaker bond that will break this cross-reference problem. This is done using *weak_ptr* instead of *shared_ptr*. We will change the *leader* pointer from *shared_ptr* to *weak_ptr* like this:

```
class worker {
public:
    int i;
    std::weak_ptr<worker> leader;    // change to weak_ptr to break the bond
    std::shared_ptr<worker> partner;
...
```

Now the connection is not so tight and when we go out of scope, and no one needs the pointers they will be auto deleted.

Many times, we need to allocate a pointer, perform some operations, and finally delete it. This is quite simple and can be done like this:

```
template<class T>
class pointer_handler {
public:
    T* ptr;
    pointer_handler(T* w=NULL) : ptr(w) {}
    ~pointer_handler() {
        if (ptr) delete ptr;
    }
};
void test_raw()
{
    pointer_handler<worker> rh(new worker(5));
}
```

This works fine and when the *pointer_handler* object goes out of scope it deletes the pointer. Simple and elegant solution we might say, but there is a problem. If we add this line of code:

```
pointer_handler<worker> rh1 = rh;
```

The program will crash. The reason is that we have two objects trying to delete the same pointer. This is a common situation in software development and C++ Standard Library has a solution. The library class is called *unique_ptr*. As the name implies there can be only one unique_ptr object handling a pointer. If we try to copy it to another unique_ptr we get a compilation error. We can only transfer ownership of the pointer.

Another benefit is that when we assign a new pointer to our unique_ptr object it deletes the old pointer before it takes over the new pointer.

```
void cause_error()
{
    throw std::runtime_error("error");
}
// unique pointer cannot be copied
// the call MUST be with reference
void use_pointer(std::unique_ptr<worker>& ptr)
{
    ptr->talk();
}
void raw_pointer(worker* ptr)
{
    std::cout << "raw:";
    ptr->talk();
}
void test_unique_ptr()
{
    std::cout << "TEST unique pointers\n";
    std::unique_ptr<worker> u(new worker(1));
    // assigning new pointer to 'u' automatically deletes old pointer
    u = std::unique_ptr<worker>(new worker(2));
    // we can get the raw worker pointer
    worker* w = u.get();
    // and use it, but it is not recommended
    raw_pointer(w);
    // this bypasses unique_ptr and will crash the program
    std::unique_ptr<worker> d(w);
    try {
        cause_error();
        std::cout << "fine!!\n";  // no exception thrown
    }
    catch (...) {  // an exception was thrown
        // unique pointers work with exceptions!
        std::cout << "error!!\n";
    }
    use_pointer(u);
    // transfer ownership to 'cp'
    std::unique_ptr<worker> cp = move(u);
    // 'u' is now invalid, so we use 'cp'
    use_pointer(cp);
}
```

As you can see, misusing unique_ptr can still lead to crashes.

Containers and iterators

We are now getting into the heart of C++ Standard Library. We are starting with *containers*. As the name implies *containers* are classes designed to hold collections of other objects. We were introduced to the concept of containers when we talked about arrays and dynamic memory management in chapters 3 and 4.

The containers manage the space required to store the elements and provide mechanisms to iterate and retrieve information.

Iterators are pointer-like objects designed to access elements inside containers. Although they are designed according to the container they access, they provide a uniform programming interface simplifying our job. We will see some of the most used containers.

Sequence containers

They store information sequentially.

array

This is an alternative to the C-style arrays we saw in chapter 3. C++ arrays are self-aware, they know their size and are more robust.

```
#include <iostream>
#include <array>

void arrays_sample()
{
    std::cout << "array sample\n";
    std::array<int, 3> ar = { 10,20,30 };
    for (auto i = 0; i < 3; ++i)
        std::cout << i << "," << ar[i] << "\n";

    std::cout << std::get<0>(ar) << "\n";
}
```

vector

Vectors can be considered as dynamic arrays. They have the ability to resize the memory they occupy when we insert elements. They occupy contiguous storage. Inserting and removing at the end is the preferable way and takes the least time. We can insert and remove it at any position as well, only this take longer.

With vectors we are introduced to iterators. We will use the *vector<type>::iterator*. This iterator *knows* the beginning and the end of the vector storage and walks up and down in the storage space.

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

void vector_sample()
{
    std::cout << "vector sample\n";
    std::vector<int> v;
    // adding data to the vector
    for (auto i = 0; i < 5; ++i)
        v.push_back(1 + 2 * i);
    // access using index
    for (auto i = 0; i < 5; ++i)
        std::cout << i << ": " << v[i] << "\n";
    // iterate elements
    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
        std::cout << *it << "\n";
}
```

deque

Double ended queue. This is a sequential container that can expand and retract from both ends. When we form queues we add elements in the back and remove them from the front. Double ended queues allow us to add and remove elements at both ends. *Deque* has a similar programming interface with vector, but we need to be careful since there is no guarantee that the memory is contiguous.

```
#include <iostream>
#include <deque>
void deque_sample() {
    std::cout << "deque sample\n";
    std::deque<int> d;
    // put some data in the container
    for (auto i = 0; i < 5; ++i){
        d.push_back(i + 5);
        d.push_front(i + 50);
    }
    // see the first element
    std::cout << "front:" << d.front() << "\n";
    // remove the first element
    d.pop_front();
    // and see the new first element
    std::cout << "front:" << d.front() << "\n";
    // iterate the container
    for (std::deque<int>::iterator it = d.begin(); it != d.end(); ++it)
        std::cout << *it << "\n";
}
```

list and forward_list

These containers implement the *linked list* data structure. *List* implements the *double linked list* in which elements point both to the previous and the next element. *Forward_list* implements the *single linked list* in which elements point only to the next element. This means that we can traverse the *forward_list* only forward as the name implies, while the *list* can be traversed both ways.

```
#include <iostream>
#include <forward_list>
#include <list>

void lists_sample()
{
    std::cout << "lists sample\n";
    std::forward_list<int> fl;
    std::list<int> l;
    // put some data in the container
    for (auto i = 0; i < 5; ++i)
    {
        // forward_list can only insert at one end
        // preferably the head for speed
        fl.push_front(i);
        // list can insert at both ends
        l.push_back(i);
    }
    // iterate the containers
    std::cout << "forward list\n";
    for (std::forward_list<int>::iterator it = fl.begin(); it != fl.end();
++it)
        std::cout << *it << "\n";
    std::cout << "list\n";
    for (std::list<int>::iterator it = l.begin(); it != l.end(); ++it)
        std::cout << *it << "\n";
}
```

Associative containers

These are containers that store data in a sorted fashion. This makes them highly efficient when we search for a certain key.

set

set is a container in which every element has to have unique value. It is this value that works as a key and makes the container fast in searches.

```
#include <iostrem>
#include <set>

void set_sample()
{
    std::cout << "set sample\n";
    std::set<int> s;
    // put some data in the container
    for (auto i = 0; i < 5; ++i)
    {
        std::cout << "inserting:" << 100 - i << "\n";
        s.insert(100 - i);
    }
    // iterate the container, note the stored order!
    std::cout << "set contents, note the order!\n";
    for (std::set<int>::iterator it = s.begin(); it != s.end(); ++it)
        std::cout << *it << "\n";
    // search for value
    std::set<int>::iterator f = s.find(98);
    if (f != s.end())
        std::cout << "found:" << *f << "\n";
    else
        std::cout << "entry not found\n";
}
```

map

This container uses a key-value pair when storing. The keys must be unique. The pair used in this container are the pairs we saw earlier in the utility library.

```
#include <iostream>
#include <map>

void map_sample()
{
    std::cout << "map sample\n";

    std::map<int, int> m;
    // put some data in the container
    for (auto i = 0; i < 5; ++i)
    {
        // create a key-value pair and insert it
        m.insert(std::pair<int, int>(i, 100 - i));
    }
    // iterate the container
    for (std::map<int, int>::iterator it = m.begin(); it != m.end(); ++it)
    {
        // print the (key, value) pair
        std::cout << it->first << ", " << it->second << "\n";
    }
    // search for a key
    std::map<int, int>::iterator f = m.find(4);
```

```
    if (f != m.end())
        std::cout << "found:" << f->first << ", " << f->second << "\n";
    else
        std::cout << "entry not found\n";
}
```

multiset

Multiset is similar to *set*. The difference being that entries do not have to be unique. Searches in multisets return subsets of the multiset instead of a single object.

```
#include <iostream>
#include <array>
#include <vector>
#include <deque>
#include <forward_list>
#include <list>
#include <set>
#include <map>

void multiset_sample()
{
    std::cout << "multiset sample\n";
    std::multiset<int> m;
    // put some data in the container
    for (auto i = 0; i < 5; ++i)
    {
        for (auto j = i + 2; j < 2 * i + 4; ++j)
        {
            m.insert(i);
        }
    }
    // iterate the container
    std::cout << "total contents\n";
    for (std::multiset<int>::iterator it = m.begin(); it != m.end(); ++it)
    {
        std::cout << *it << "\n";
    }
    std::cout << "all the 3s\n";
    for (std::multiset<int>::iterator it = m.lower_bound(3); it !=
m.upper_bound(3); ++it)
        std::cout << *it << "\n";
}
```

multimap

Multimap is a map with that does not require the keys to be unique. So, when we are searching for a key we get the subset of the stored values that share the same key value.

```
void multimap_sample()
{
    std::cout << "multimap sample\n";
    std::multimap<int, int> m;
    // put some data in the container
    int id = 1;
    for (auto i = 0; i < 5; ++i)
    {
        for (auto j = i + 2; j < 2 * i + 4; ++j)
        {
            m.insert(std::pair<int, int>(i, id));
            ++id;
        }
    }
    // iterate the container
    std::cout << "total contents\n";
    for (std::multimap<int, int>::iterator it = m.begin(); it != m.end(); ++it)
    {
        // print the (key, value) pair
        std::cout << it->first << ", " << it->second << "\n";
    }
    std::cout << "all the 3s\n";
    for (std::multimap<int, int>::iterator it=m.lower_bound(3);
it!=m.upper_bound(3); ++it)
        std::cout << it->first << ", " << it->second << "\n";
}
```

Unordered associative containers

These containers implement unsorted structure that can be quickly searched. Due to their unsorted nature, they are faster to insert and remove but, they are a little slower in general that *associative containers*. These containers are:

- Containers with unique keys
    - *unordered_set*: a set of hashed keys
    - *unordered_map*: a collection of key-value pairs, with hashed keys
- Containers with multiple entries of the same key
    - *unordered_multiset*: a set of possibly repeated hashed keys
    - *unordered_multimap*: a collection of key-value pairs, with possibly repeated hashed keys

These containers offer the same functionality as their ordered counterparts.

Container adapters

These containers are based on *sequence containers* and provide us with special interfaces.

stack

This adapter is based on *queue* container and implements a Last In First Out (LIFO) data structure. It is actually a pile of data that you only add at the top and you can only retrieve the top object.

queue

This is a real queue. It implements the First In First Out (FIFO) data structure. The first to enter is the first to be served.

priority_queue

This queue prioritizes its contents so that we always get the largest, no matter in which order we insert them into the container. Internally it stores data in a *vector*, and we can define our own comparison operator so that we can sort the data in any way we want. Defining our own comparison is very helpful if we want to store custom classes instead of built-in types.

```
void priority_queue_sample()
{
    std::cout << "priority_queue sample\n";
    // instead of the default ordering we set the greater operator
    std::priority_queue<int, std::vector<int>, std::greater<int> > m;
    // put some data in the queue
    for (int i : {1, 8, 5, 6, 3, 4, 0, 9, 7, 2})
    {
        m.push(i);
    }
    // look at the top element
    std::cout << "top element before:" << m.top() << "\n";
    // insert an even smaller
    m.push(-3);
    // and check again
    std::cout << "top element after:" << m.top() << "\n";
}
```

Strings

So far we are treating text using the C language *char** pointer notation of *char[]* array notation. Strings are indeed character arrays in computer memory, but C++ has a more sophisticated approach.

Part of the Standard Library is the *std::string* class. This class wraps around the primitive pointer giving us an easier and more robust interface. String classes can be used as template parameters, implement operators for easier handling in the code, they can be

safely accessed without looking for *NULL* termination character. The memory is dynamically allocated, extended, and released as needed and we do not have to worry about it.

```
void string_sample()
{
    std::string s1("this is ");
    std::string s2("a string");
    std::string s3(s1 + s2);
    std::cout << s3 << "\n";

    // use the C printf function to print the string
    // this function requires a char* pointer
    printf("%s\n", s3.c_str());

    // append some text
    s3 += " for testing";
    std::cout << s3 << "\n";
    // get a substring from position 2 and of length 6
    std::cout << s3.substr(2, 6) << "\n";
    // replace from char 2 to 4 with the string 'lala'
    s3.replace(2, 4, "lala");
    std::cout << s3 << "\n";
    // find 'str' and print from there to the end of string
    size_t p = s3.find("str");
    if (p != std::string::npos) std::cout << s3.substr(p) << "\n";
    // use array notation to access characters
    std::cout << "sixth character=" << s3[5] << "\n";
}
```

Function objects

In C++ the () operator is the *function call* operator. Any object that implements this operator is a *function object* or a *functor*.

Function objects are used to improve function pointers we saw in chapter 5. With function pointers we could only perform a function call. Function objects are a lot more flexible because they store their state for consecutive calls. Another advantage is that they are types and so we can use them as template parameters.

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

// our compare function object
template<class T>
class compare {
public:
    bool operator()(T l, T r) {
        // return the comparison result
```

```
        return l < r;
    }
};

// simple templated function to find an extreme value
// in a vector of T using the function object above
// if no compare type is given <int> is assumed
template<class T, class comp=compare<int>>
T find_extreme(std::vector<T>& v)
{
    comp c;   // create the compare object
    T ex = v[0];  // the first is the extreme
    // iterate vector
    for (auto it = v.begin(); it != v.end(); ++it)
    {
        // if a new extreme is found, store it
        if (c(ex, *it))
            ex = *it;
    }
    // return extreme
    return ex;
}

void functor_sample()
{
    std::cout << "function object sample\n";
    std::vector<int> v;
    for (int i : {1, 8, 5, 6, 3, 4, 0, 9, 7, 2})
        v.push_back(i);
    // first run with default compare template
    auto r = find_extreme<int>(v);
    std::cout << "extreme int value:" << r << "\n";

    std::vector<double> d;
    for (double i : {1.2, 8.3, 5.4, 6.5, 3.6, 4.7, 0.8, 9.9, 7.1, 2.2})
        d.push_back(i);
    // for second run we request compare for doubles
    auto q = find_extreme<double, compare<double>>(d);
    std::cout << "extreme double value:" << q << "\n";
}
```

This is a problem we could solve using *priority queues*. Had this been a real-world application we might have done just that, but for the shake of this example we did it with a simple template function.

Lambdas

Lambda expressions or *lambda functions* are *unnamed objects* that can be called like functions. They allow us to create inline functions with code that is not meant for reuse.

The syntax of a lambda function is

```
[capture_clause] (parameters) -> return_type
{
    statement(s)
}
```

The most powerful characteristic of lambda functions over normal functions is that they can have access to the local variables of the enclosing scope. That is declared by the *capture_clause*. It can be by value, by reference or mixed capture, other variables by value and other by reference.

Here is a syntax for each case:

```
[=]  : capture all variables by value
[&]  : capture all variables by reference
[x, &y] : capture x by value and y by reference
```

The *parameters* are like ordinary function parameters. The *return_type* is usually determined by the compiler automatically. It is only required when the function is ambiguous, and the compiler cannot make out the return type. Finally, the code segment is the function code of the lambda closure.

```
// this function iterates the vector and
// counts how many times the 'func' returns true
template<class T, class func>
int count_objects(std::vector<T>& v, func f)
{
    int count = 0;
    for (auto it = v.begin(); it != v.end(); ++it)
    {
        if (f(*it))
            ++count;
    }
    return count;
}

void labda_sample()
{
    // basic lambda, we create an inline function
    auto greet = []() {std::cout << "lambda sample\n"; };
    greet();

    std::vector<int> v;
    for (int i : {1, 8, 3, 4, 0, 9, 7, 2, 1, 3, 5, 6, 3, 4, 7, 2, 1, 8, 5, 6,
3, 9, 7, 2})
        v.push_back(i);
    // count how many '3's are in the vector
    auto se = [](int i) { return i == 3; };
    std::cout << "found:" << count_objects(v, se) << "\n";

    // count how many numbers in the vector are between 3 and 7
```

```
    // we can write the code inline, but it is not so readable
    std::cout << "found:" << count_objects(v, [](int i) { return i >= 3 && i
<= 7; }) << "\n";

    int x = 2;
    // here we define a lambda function that returns 'bool'
    // the return type in this example is optional
    // it is also accessing access the local variable by value
    auto l = [=](int y) ->bool { return y * x; };
    std::cout << "result:" << l(3) << "\n";

    // here we are accessing x by reference
    auto lr = [&](int y) {++x; return y * x; };
    std::cout << "result:" << x << ", " << lr(4) << "\n";
}
```

Algorithms

The C++ Standard Library contains a set of useful *algorithms*. They are designed to assist in common problems such as sorting and searching. Since these are very common operations we need to perform on data stored in containers they are implemented in the library. This gives us the benefit of standard code that is highly optimized and debugged. We can focus on our problems and be sure that the tools we use are of the highest quality.

What sets the C++ algorithms apart from other language implementations is that they operate on iterators and not on containers. As long as your data storage implements iterators correctly the algorithms can be used.

The only disadvantage is that because they operate on iterators they have their vulnerabilities, which we can easily overcome with some careful programming.

Analyzing all the algorithms in the library is beyond the scope of this writing. We will see some of the most common algorithms we might need in the development of the simple game engine we will create in this book.

The algorithms library relies heavily on function objects. That is our chance to inject our logic and modify the behavior of the algorithms.

for_each

This algorithm iterates the container starting from the *begin* iterator to the *end* iterator given, applying the function object we pass.

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

class point {
public:
    int x, y;
    point(int _x = 0, int _y = 0) :x(_x), y(_y) {}
    ~point(){}
};

void foreach_sample()
{
    std::cout << "first sample\n";

    std::vector<point> v;
    for (auto i = 0; i < 5; ++i)
        v.push_back(point(i + 1, i + 2));

    // from the first point to the last apply the lambda function
    // move all the points by a fixed amount
    std::for_each(v.begin(), v.end(), [](point& p) {p.x += 2; p.y += 7;});
    // print the coordinates of all the points
    std::for_each(v.begin(), v.end(), [](point& p) {std::cout << p.x << ", "
<< p.y << "\n";});
}
```

We could have written a function to iterate over the objects, but it would take some time to make it this general and versatile.

Sorting algorithms

The first group of algorithms we are going to see are the sorting algorithms. Keeping our objects sorted based on some key criteria can significantly reduce processing speed. We can rely on sorting to avoid unnecessary lengthy calculations.

sort

The first and most basic sorting algorithm in the library. It comes in two flavors. The first compares two objects using the *less (<)* operator, and the second uses a *compare function object* we provide

```
template <class Iterator>
void sort(Iterator first, Iterator last);

template <class Iterator, class Compare>
void sort(Iterator first, Iterator last, Compare comp);
```

Here is a sample

```
void sort_sample()
{
    std::cout << "sort sample\n";
    std::vector<std::pair<int, std::string>> v;
    int count = 1;
    for (int i : {1, 8, 5, 6, 3, 4, 0, 9, 7, 2})
    {
        std::string name = "object " + std::to_string(count++);
        v.push_back(std::pair<int, std::string>(i, name));
    }

    std::cout << "unsorted vector\n";
    std::for_each(v.begin(), v.end(), [](std::pair<int, std::string>& p)
        {std::cout << p.first << ", " << p.second << "\n"; });
    std::cout << "sorted by their integer\n";
    std::sort(v.begin(), v.end());
    std::for_each(v.begin(), v.end(), [](std::pair<int, std::string>& p)
        {std::cout << p.first << ", " << p.second << "\n"; });
    std::cout << "sorted by their string\n";
    std::sort(v.begin(), v.end(), [](std::pair<int, std::string>& l,
std::pair<int, std::string>& r)
        {return l.second > r.second; });
    std::for_each(v.begin(), v.end(), [](std::pair<int, std::string>& p)
        {std::cout << p.first << ", " << p.second << "\n"; });
}
```

stable_sort

This is almost the same as *sort*. The only difference is that if two elements are equivalent it does not swap them.

partial_sort

This function can sort only a part of the container.

```
template <class Iterator>
void partial_sort(Iterator first, Iterator middle, Iterator last);

template <class Iterator, class Compare>
void partial_sort(Iterator first, Iterator middle, Iterator last, Compare
comp);
```

It affects the range *[first, last)* iterator. It puts in the range up to *middle* the smallest elements in ascending order, and from *middle* to *last* the remaining elements keep their relative order.

The following sample shows how *partial_sort* works

```
void partial_sort_sample()
{
    std::cout << "partial_sort sample\n";
    std::vector<std::pair<int, std::string>> v;
    int count = 1;
    for (int i : {10, 8, 5, 6, 3, 4, 0, 9, 7, 2, 11, 13, 19, 17, 16, 14, 12,
15, 18, 1})
    {
        std::string name = "object " + std::to_string(count++);
        v.push_back(std::pair<int, std::string>(i, name));
    }
    std::for_each(v.begin(), v.end(), [](std::pair<int, std::string>& p)
        {std::cout << p.first << ", " << p.second << "\n"; });
    std::cout << "---------------\n";
    std::partial_sort(v.begin(), v.begin()+7, v.begin()+12);
    std::for_each(v.begin(), v.end(), [](std::pair<int, std::string>& p)
        {std::cout << p.first << ", " << p.second << "\n"; });
}
```

Searching algorithms

We were introduced to search when we talked about containers and iterators. Searching and retrieving information is one of the basic functions we do with containers. Here we will look at some fundamental and common search algorithms.

find

find searches a specified range within a container for a specific value. If we are searching in a container of custom types we must define the '==' operator in our class.

```
class foo {
public:
    int x;
    std::string s;
    foo(int xx = 0, std::string ss = "") :x(xx), s(ss) {
    }
    // overloaded operators to be used by std::find
    // find integer
    bool operator==(int i) {
        if (i == x)
            return true;
        return false;
    }
    // find matching object
    bool operator==(const foo& f) {
        if (f.x == x && f.s == s)
            return true;
        return false;
    }
```

```
    // find string
    bool operator==(const std::string& _s) {
        if (_s == s)
            return true;
        return false;
    }
};

void find_sample()
{
    std::cout << "find sample\n";
    std::vector<foo> v;
    int count = 1;
    for (int i : {10, 8, 5, 6, 3, 4, 0, 9, 7, 2, 11, 13, 19, 17, 16, 14, 12,
15, 18, 1})
    {
        std::string name = "object " + std::to_string(count++);
        v.push_back(foo(i, name));
    }

    auto p = std::find(v.begin(), v.end(), "object 4");
    if (p != v.end())
        std::cout << "found:" << p->x << ", " << p->s << "\n";
    else
        std::cout << "not found\n";
}
```

find_if & find_if_not

These functions are complementing *std::find*. Instead of using the *'operator=='* of the object stored in the container they take a *function object* that returns a Boolean value. *Find_if* returns the first 'true' object, and *find_if_not* the first 'false'. Here is an example:

```
void find_if_sample()
{
    std::cout << "find_if sample\n";
    std::vector<foo> v;
    int count = 1;
    for (int i : {10, 8, 5, 6, 3, 4, 0, 9, 7, 2, 11, 13, 19, 17, 16, 14, 12,
15, 18, 1})
    {
        std::string name = "object " + std::to_string(count++);
        v.push_back(foo(i, name));
    }

    auto p = std::find_if(v.begin(), v.end(), [](const foo& f)
        {return f.s == "object 8"; });
    if (p != v.end())
        std::cout << "find_if:" << p->x << ", " << p->s << "\n";
    else
        std::cout << "not found\n";
```

```
    auto p1 = std::find_if_not(v.begin(), v.end(), [](const foo& f)
        {return f.s == "object 8"; });
    if (p1 != v.end())
        std::cout << "find_if_not:" << p1->x << ", " << p1->s << "\n";
    else
        std::cout << "not found\n";
}
```

find_first_of

Search a range and return the first occurrence of any element of another range. To make it clear, search the people of a town and get the first one that is called either ‘Jack’, ‘Joe’, or ‘Jim’.

```
void find_first_of_sample()
{
    std::cout << "find_first_of sample\n";
    // our data
    std::vector<int> v{ 10, 8, 5, 6, 3, 4, 0, 9, 7, 2, 11, 13, 19, 17, 16, 14,
12, 15, 18, 1 };
    // what we are looking for
    std::vector<int> s{ 1,2,3 };

    // find the first occurrence of any element of 's' in 'v'
    auto f = std::find_first_of(v.begin(), v.end(), s.begin(), s.end());
    if (f != v.end())
        std::cout << "find_first_of:" << *f << "\n";
    else
        std::cout << "not found\n";
}
```

lower_bound & upper_bound

They return iterators to the first and last occurrence of a certain value, respectively. For the functions to operate the range must be fully sorted.

```
void bounds_sample()
{
    std::vector<int> v{1,2,3,4,5,5,5,6,7,8,9,10};
    std::sort(v.begin(), v.end());
    auto s = std::lower_bound(v.begin(), v.end(), 5);
    auto e = std::upper_bound(v.begin(), v.end(), 5);
    for (auto p = s; p != e; ++p)
    {
        std::cout << *p << "\n";
    }
}
```

Copy and move operations

These are data management operations we can do to keep things tidy in our programs

copy / copy_if

These functions copy a range from one container to another. The first performs an unconditional copy and the later calls a predicate to determine if it should copy or not.

```
void copy_sample()
{
    std::cout << "copy sample\n";
    std::vector<int> v{ 10, 8, 5, 6, 3, 4, 0, 9, 5, 2, 11, 13, 19, 17, 16, 5,
12, 15, 18, 1 };
    std::vector<int> l;

    // copy 4 items in l
    std::copy(v.begin(), v.begin()+4, std::back_inserter(l));
    // print the result
    std::for_each(l.begin(), l.end(), [](int p) {std::cout << p << "\n"; });
    std::cout << "-----------\n";
    // copy the odd numbers to another vector
    std::vector<int> odds;
    std::copy_if(v.begin(), v.end(), std::back_inserter(odds),
                 [](int i) {return i % 2; });
    std::for_each(odds.begin(), odds.end(),
                 [](int p) {std::cout << p << "\n"; });
}
```

remove / remove_if

These functions remove either a value or elements that fulfill certain criteria from a range. They pack the data to the beginning of the container and return the iterator after the last element to keep. So, it is our responsibility to clear the ‘bad’ elements.

```
void remove_sample()
{
    std::cout << "remove sample\n";
    std::vector<int> v{ 10, 8, 5, 6, 3, 4, 0, 9, 5, 2, 11, 13, 19, 17, 16, 5,
12, 15, 18, 1 };
    std::for_each(v.begin(), v.end(), [](int p) {std::cout << p << " "; });
    std::cout << "\n";

    // remove 5s
    auto i1 = std::remove(v.begin(), v.end(), 5);
    // clear the unwanted elements
    v.erase(i1, v.end());
    // print the result
    std::for_each(v.begin(), v.end(), [](int p) {std::cout << p << " "; });
```

```
    std::cout << "\n";

    // remove odds
    auto i2 = std::remove_if(v.begin(), v.end(), [](int i) {return i%2; });
    v.erase(i2, v.end());
    std::for_each(v.begin(), v.end(), [](int p) {std::cout << p << " "; });
    std::cout << "\n";
}
```

merge

This function merges two sorted ranges into one. Objects stored in the ranges must provide the *less* (<) operator or we must provide a *compare function object* for sorting.

```
void merge_sample()
{
    std::cout << "merge sample\n";
    std::vector<int> v{ 1,2,3,4,5,6 };
    std::vector<int> w{ 10,11,12,13,14 };
    std::vector<int> r;

    std::merge(v.begin(), v.end(), w.begin(), w.end(), std::back_inserter(r));
    std::for_each(r.begin(), r.end(), [](int p) {std::cout << p << " "; });
    std::cout << "\n";
}
```

min / max

Return the minimum or the maximum of a list of objects. A compare function can be provided.

```
void minmax_sample()
{
    std::cout << "min/max sample\n";
    // standard min/max for two integers
    std::cout << std::min(3, 4) << "\n";
    // an error in the compare function can make max to return min!
    std::cout << std::max({3,4,5,6,2},
              [](int i, int j) {return i > j; }) << "\n";
}
```

Here ends our introduction to algorithms. They are so many that writing a complete list is far beyond the purpose behind this book.

Summary

We covered the basics of the C++ standard Library. I/O was deliberately left out. We will take a more detailed approach in the next chapter. In this chapter we covered

- the utility library
- containers and iterators
- strings
- function objects
- lambdas
- algorithms

Chapter 11: Input / Output

Up until now we have seen some I/O in the form of *std::cout* output stream that we used to print information on the screen.

We need a lot more than that. We need to store the state of our program for later retrieval, or read information someone else created, like an image.

The C++ Standard Library makes all this very easy. It has classes that do more than reading from the keyboard and writing to the screen. We can use it to store or retrieve any kind of information either sequentially or in random order.

Streams

In C++ we call the I/O mechanism *stream* because of the stream of information between our program the I/O. so we use *stream* classes for input and output operations between our program and files or devices.

So far we have seen *std::cin* and *std::cout* that read the keyboard and print on the screen, respectively. They are predefined *objects* that live in the library:

```
#include <iostream>

int main()
{
    std::string s;
    std::cout << "please type your name:";
    std::cin >> s;
    std::cout << "Hello " << s << "\n";
    return 0;
}
```

The *stream operator* (<<) lets us concatenate one object after the other. This is done with *operator overloading*. The stream operator is predefined in the library for all the basic types as well as for stream able objects like strings.

The basic syntax for I/O in C++ is the same because all the files and devices are declared as streams.

String streams

This class brings together strings and streams. It works both for input and for output. We can output to this stream like we do with *cout*, and then get the string representation of our output, or read a string as if we are reading from the keyboard.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <sstream>

void strinstream_sample()
{
    // initialize the stream with a string
    std::stringstream s("this is a string with several words");
    std::string w;
    std::vector<std::string> v;

    // while we are reading words
    while (s >> w)
    {
        // we put them in a vector
        v.push_back(w);
    }
    // this will be our output stream
    std::stringstream t;
    // output the words followed by a hyphen
    std::for_each(v.begin(), v.end(),
       [&t, &w](std::string& s) {std::cout << s << " "; t << s << "-"; });
    // display the result
    std::cout << "\n" << t.str();
}
```

File streams

Now that we have the basic understanding of how the streams work it is time to learn about files and file streams.

Files

By definition computer files are resources used to store information either to share or for later retrieval. Files are stored in non-volatile computer resources. In the early days of computing these included punched cards or magnetic tapes, while now we use hard disks, solid state disks and other ultra-fast and highly dense media.

We use files to store images, text, the state of our program and any kind of data we generate working with our computers. We all know that information becomes knowledge only if we store it for later use, otherwise it gets lost. The knowledge generated is multiplied if it is shared so another vital operation we can perform with files is to share them, either by exchanging media, as was done in the past, or by sending them over the internet as we do today.

File types

Files are divided in two main categories. *Text files* and *binary files*.

As the name implies *text files* contain text. We need the string representation of a number to save it into a text file.

Binary files on the other hand contain raw memory contents. Text is transferred to the file by copying the memory that contains it as we do with numbers.

Whatever the type of the file, the data in it must be organized. Unordered information is just noise. We can use markers to mark segments of the file. For instance, the start of each line of an image in an image file, the number of pixels in each line and the number of lines, are markers that can be used. These conventions should be documented as well, and the documentation should be updated every time we need to make any changes.

This file organization is what we call *file format*.

File operations

The next thing we are going to address is what we can do with files. What are the operations we can perform?

- We can create a file
- Write data to the file
- Read data from the file
- Delete the file
- Copy or move the file to another location

Using files

In order to use files in our program we need the *<fstream>* library, which is part of the C++ Standard Library. This library defines the three new data types we need to work with files.

ofstream	This is the output file stream. It is used to create files and write information.
ifstream	This is the input file stream. It is used to read information from existing files.
fstream	This is a general file stream. It has the capabilities of both ofstream and ifstream and can be used for any operation.

Open a file

The first thing we must do to use a file is to open it. To open a file for writing we need either *ifstrem* or *fstream*, whereas to open it for reading we need *ofstream* or *fstream*.

The file can is opened using the *open* function:

```
std::fstream o;
o.open("some file.txt", std::ios::mode);
```

The arguments are the file name and the *mode* of the file.

ios::app	Open the file for appending data
ios::ate	Open the file for appending data and move rea/write control to the end
ios::in	Open the file for input
ios::out	Open the file for output
ios::trunc	Open the file and if it exists truncate its contents
Ios::binary	Open the file in binary mode, default is text mode

Closing a file

When we are done with the file it is a good practice to close it. The library takes care of this automatically when the stream object goes out of scope but depending on the complexity of our code this might take some time, so it is better if we take care of it ourselves.

```
o.close();
```

Writing text to a file

The fact that we have a stream makes this operation very easy. We have seen how to write to the standard output stream. Writing to a file stream is the same. Just replace the stream object with our file stream object.

```
void filewrite_sample()
{
    std::ofstream o;
    // ofstream is by default for output
    o.open("some file.txt", std::ios::out);
    o << "some text";
    o.close();
    // reopen and append some text
    o.open("some file.txt", std::ios::app);
    o << "some more text";
    o.close();
}
```

Reading text from a file

Reading text from a file is like reading the keyboard. We can use the stream operator and we will read the file *word by word*. The input stream breaks at white space. We can overcome this if we use the *getline* function which reads to the end of the line.

```
void fileread_sample()
{
    std::ifstream is;
    is.open("some file.txt", std::ios::in);

    std::string s;
    // using stream operator reads word by word
    // while (is >> s)
    // using getline reads line by line
    while (getline(is, s))
    {
        std::cout << s << "\n";
    }
    is.close();
}
```

Writing binary files

Binary files are not so straight forward. Stream operators output text and text representation of numbers. Human readable information is the default in the library. Whatever we see on the screen goes into the file. This means that we will have to use other functionality within the library to write to a binary file.

Binary vs text

What is the difference between binary and text data and why do we need to do anything?

The basic component of the way information is stored in the computer is the *binary digit* or *bit*. A bit can have two values, either 1 or 0. Eight bits form one *byte*. Bytes can store a number from -128 to 127 or 0 to 255 depending on if we want signed or unsigned numbers.

By convention we use an unsigned byte to represent text. The value 48 is for 0, 49 for 1, 65 for A, 66 for B and so on. This is called the ASCII table of characters.

When we save ‘1’ in a text file, we write the number 49, whereas in a binary file we just write 1. In all other operations we use the value 1 and not 49 when we talk about 1.

This is the basic difference between text and binary data and files for that matter. A convention we have to live with and as software developers we have to deal with.

Binary output

Now that we have cleared what we want to do is the time to see how we can do it.

The first solution is the function *write*, which is member of the stream object. Contrary to the stream operator that coverts from binary to text, this function outputs raw data from the computer memory to the file. It can be used to output any primitive type such as *int, double, char* and the rest.

```
void write_bin_sample()
{
    std::ofstream o;
    // open the file for binary output
    o.open("some file.dat", std::ios::out | std::ios::binary);
    int i = 5;
    // write an integer
    o.write((char*)&i, sizeof(int));
    // save a text string
    std::string s("some text");
    // first save its length
    i = s.length();
    o.write((char*)&i, sizeof(int));
    // and then the contents, c_str returns the char* within the string
    o.write((char*)s.c_str(), i);
    o.close();
}
```

The other option we have is to create our own output stream class that overrides the stream operators so our program will not have to change. Here is a possible implementation:

```
// our custom output stream class
class my_ostream : public std::ofstream {
public:
    my_ostream& operator<<(int v) {
        write((char*)&v, sizeof(int));
        return *this;
    }
    my_ostream& operator<<(char v) {
        write((char*)&v, sizeof(char));
        return *this;
    }
    my_ostream& operator<<(const std::string& v) {
        int l = v.length();
        write((char*)&l, sizeof(int));
        for (auto i=0; i<l; ++i)
            write((char*)&v[i], sizeof(char));
        return *this;
    }
};
```

```
// now we can use the stream operators
void write_bin_sample()
{
    my_ostream o;
    o.open("some file.dat", std::ios::out | std::ios::binary);
    // this will write 1 and 51 (ASCII for 3)
    o << 1 << '3';
    // and this will write the text
    o << std::string("some text");
    o.close();
}
```

Reading binary files

When reading from binary files we have to rely on functions instead of stream operators again. As you can guess the function to use for reading is *read*. Here we read the file we created before.

```
void read_binary_sample()
{
    std::ifstream ifs;
    ifs.open("some file.dat", std::ios::in | std::ios::binary);
    // read the integer
    int i;
    ifs.read((char*)&i, sizeof(int));
    // read the character
    char c;
    ifs.read((char*)&c, sizeof(char));
    // read the string
    int l;   // first read the length
    ifs.read((char*)&l, sizeof(int));
    // allocate memory (+1 for NULL terminator)
    char* txt = new char[l+1];
    // read the text
    ifs.read(txt, l*sizeof(char));
    txt[l] = '\0';  // append NULL terminator
    std::string s(txt);   // and create a string
    delete []txt;    // release text memory
    ifs.close();
    // and view the results
    std::cout << "i=" << i << "\n";
    std::cout << "c=" << c << "\n";
    std::cout << "s=" << s << "\n";
}
```

Obviously, we can create a custom output stream class that implements the stream operators. That would significantly improve our code especially when reading strings, since it would hide all things we have to do to read and write them.

A closer look at the write and read samples shows what we mean with file format. The fact that we saved in strict order an integer, a character and a string, and the way we saved the string in particular, is characteristic. Following the same strategy when writing and reading a file regardless of if it is a text or a binary file is like the two are speaking the same language.

Summary

On this chapter we got some idea about

- Streams
- String streams
- File streams
- File types and file operations
- Opening and closing files
- Using text files
- Using binary files

Chapter 12: Error handling

Errors in programming are the standard. However careful we may be there comes a time that our code encounters situations it cannot handle. Some errors occur because we forgot to check ad divided by zero for example, and some due to user errors, i.e., attempting to open a file he had no permission to open.

These errors might lead to an abnormal program termination. This is very frustrating for the user and eventually bad for our programs.

C++ has a built-in mechanism to resume operation when an error occurs. This in conjunction with some classic programming techniques that we are going to see in this chapter will help us create more robust programs.

Avoid errors

We must accept the fact that we cannot eliminate errors altogether. As humans we are susceptible to errors. But we have an advantage. We can learn from our mistakes, and the mistakes of those before.

Many books about techniques to improve the quality of our code have been written over the years. Here are some simple things we can do to improve our code.

Reuse code

Break your code down to small and flexible pieces you can reuse in many projects, or within the same project. Take a look at the *trinomial* function we saw in chapter 1.

```
double trinomial(double a, double b, double c, double x)
{
    double result = a * x * x + b * x + c;
    return result;
}
```

This is a small and elegant function you can call from numerous points in the code to perform the specific calculation.

Reusing code leads to better code over time because in these simple functions are easier to spot errors and the most we use the code the more error come to surface, and we can fix them.

Validate your data

Always check if your data is valid. This is something you should do as early in the process as possible. For example, check your data before passing them to a function like the function above. Check the divider and avoid dividing by zero. Check a pointer and do not read or write using NULL pointers.

If you can make sure that the data the user entered are valid and within the limits you expect before going on with using them in your calculations. This has double benefit. You avoid crashes and you can eliminate checks deep in your code making it faster.

Use reliable external libraries

This follows the previous tip. It is the pinnacle of code reusability. These libraries have undergone so much testing over the time that they can be the last place to search if something goes wrong.

The most characteristic example is the Standard Library.

Testing your code

Develop tests for your code as you develop your code. Compile your code as you develop to catch all the typing and syntax errors before they become too many and start giving you a hard time.

Run your tests every time you complete a step and make sure that everything works as expected up to this point.

Developing with tests in mind is quite hard but it pays back in the end.

Exceptions

So far we have done everything we could to avoid our mistakes. There are situation though where the user will try something that leads to an error, or we might encounter a media failure while processing, some network latency for example.

These exceptional situations raise *exceptions*. C++ has a built-in mechanism to catch these exceptions. The Standard Library on the other hand gives us the programming interface to handle them gracefully.

Unfortunately, not all the errors generate exceptions. Division by zero and accessing NULL pointers are among the errors we have to catch ourselves. This is the standard as defined by the C++ ANSI committee.

Try / catch

This is the C++ way of catching the errors that raise exceptions before they bring down our program. *Try* and *catch* are language keywords and the mark the block of code that might generate the exception and the block that is the response to that exception.

If an exception is raised anywhere within the *try* block, the code jumps to the *catch* block. There we have the chance to reset the state of our program and resume execution.

```
void simple_exception()
{
    std::cout << "simple exception sample\n";
    // we will use this to generate an exception
    int i=-1;
    try
    {
        // bad allocation exception
        char* buf = new char[i];
        // since the previous line has an error
        // the next line will not be executed
        std::cout << "this should not be seen\n";
    }
    catch(...)  // catch all exceptions
    {
        // the code resumes here after the exception
        std::cout << "error!!!\n";
    }
    // exception or not the code continues here
    std::cout << "function terminates\n";
}
```

The *new* operator generates an exception if we pass it an invalid parameter, as we do here. When the error occurs the code jump into the *catch* block. In any case the final statement is always executed, indicating that regardless of the error that occurred the function terminated without problems.

This implementation of the *try/catch* statement catches all the exceptions. Furthermore, by catching the exception using the *ellipsis* operator (…) we cannot tell what really happened.

So, we need to improve our *catch* statement. The Standard Library contains a specific class for exceptions '*std::exception*'. It is the base class for all specific exceptions, and we are going to use it. Here is how the *catch* statement will be like:

```
catch(const std::exception& e)
{
    // the code resumes here after the exception
    std::cout << e.what() << "\n";
}
```

It is still general and catches all the exceptions, only this time it has access to the exception that was raised, and we can get some feedback.

Catching specific exceptions

There are times when our response to exceptions depends on the type of the specific exception. In the previous example we try to allocate -1 bytes of memory. This is obviously impossible, so the *new* operator raises a *'bad_array_new_length'* exception.

We would like to catch this specific case because it reveals an error in some part of our program. This is how we will modify our code to handle this exception separately:

```
// catch the bad_array_new_length exception
catch(const std::bad_array_new_length& e)
{
    // the code resumes here after the bad_array_new_length exception
    std::cout << "specific exception:" << e.what() << "\n";
}
// catch all other exceptions
catch (const std::exception& e)
{
    // the code resumes here after any other exception
    std::cout << "general exception:" << e.what() << "\n";
}
```

We can line up the exceptions we want and catch them separately one by one. When we are done with them we can catch all the remaining by catching the base exception type.

Throwing exceptions

In the previous section we saw how to write safe code. The question is how to respond to a situation when something goes wrong. How can we unwind the code in a bad situation?

Exceptions allow us to do just that. We can throw an exception and automatically return control of the program to the first *catch* statement that is designed to catch it.

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

// our custom exception type
class my_exception: public std::exception {
public:
    my_exception(const char* msg= "this is a custom exception")
:std::exception(msg) {

    }
};
```

```
void throw_some_exception()
{
    throw my_exception();
}

void throw_sample()
{
    std::cout << "throw exception sample\n";
    try
    {
        throw_some_exception();
        // since the previous call raises an exception
        // the next line will not be executed
        std::cout << "this should not be seen\n";
    }
    // catch our custom exception
    catch (my_exception& e)
    {
        std::cout << e.what() << "\n";
    }
    // ignore all other exceptions

    // if the code continues here the all is ok
    std::cout << "throw exception terminates\n";
}
```

A well-designed exception handling system along with custom exception types we can create a safety net in our program so that if anything goes wrong we can bure that it will not affect the general functionality. The user will always be happy using a product that does not let him down, and in the event that something goes wrong we will have enough feedback to fix it.

First we created our custom exception type derived from the basic exception type in the library. Then we created the *catch* statement that is supposed to catch this particular exception. In the *try* statement we place the call to the function we think might raise the exception.

Note that we are only interested in our own custom exception. All the other types will be handles higher in the call hierarchy like this:

```
void higher_sample()
{
    try {
        throw_sample();
    }
    catch (const std::exception& e)    {
        // the code resumes here after any other exception
        std::cout << "general exception:" << e.what() << "\n";
    }
}
```

This is a function that calls the sample function. If we throw our custom exception the sample function will catch it. Any other exception will be caught by this higher function.

Summary

Exceptions can serve two roles. The help us get by from situations that may cause problems, and also guide our code out of dark dungeons.

The main topics of this chapter are

- Errors and how to avoid them
- Exceptions and how to catch them
- Throwing our own exceptions
- Using exception for our benefit

Chapter 13: The Preprocessor

A feature inherited for the C programming is the *preprocessor*. As the name implies the preprocessor is not a part of the C++ compiler. It is a separate program that opens the source file and 'prepares' it for the compiler. It actually substitutes parts of the code and adds instructions for the compiler.

Preprocessor commands

Preprocessor commands begin with a hash symbol (#). This character must be the first of the line, but not necessarily on the first column. Then follows the command keyword. Blanc space between the hash symbol and the keyword is allowed. The unwritten law of programming though says that the hash symbol should be at the first column and there should be no space between it and the keyword. This is the list of the preprocessor commands:

#include: this command is used to *include* another file. The file substitutes the line of the command. We have used it several times to include *header files*, containing the definitions of the functions and types we wanted to use.

```
#include <windows.h>
```

#define: this one defines a new *macro*. Whenever the preprocessor encounters the macro within our code, replaces it with the definition. Take a look at the code:

```
#define PI 3.1415926
double area = PI * radius * radius;
```

Early C did not support the keyword *const* for variables. This definition creates a pseudo constant *PI* we can use in our calculations.

Another use of the *#define* command is a pseudo template function:

```
#define AREA_OF_CIRCLE(radius) PI*radius*radius
double area = AREA_OF_CIRCLE(10);
```

Instead of writing the equation every time or having the function call overhead we create this simple definition.

#undef: undefines a previously defined macro.

```
#undef AREA_OF_CIRCLE
```

#if: check if a condition is met. If the condition is met, the code within the *#if* block is parsed by the preprocessor. In the following example if we are compiling for Windows95 or newer we can include some extra features:

```
#if (WINVER >= 0x0400)
#include "enhanced_features.h"
#endif
```

#ifdef: this returns true if a macro is defined

#ifndef: this returns true if a macro is NOT defined

#else: the alternate path when the above checks fail

#elif: the alternate path with a twist, it is the *else if*

#endif: closes the conditional block

```
#ifdef _WINDOWS_
#define DEBUG_PRINT OutputDebugString
#elif defined (_LINUX_)
#define DEBUG_PRINT linux_printf
#else
#define DEBUG_PRINT printf
#endif

#ifndef _DOUBLE_PRESICION_
#define my_float float
#else
#define my_float double
#endif
```

#error: generates error which stops compilation and print error message

```
#ifndef _WINDOWS_
#error _WINDOWS_ is not defined, include windows.h
#endif
```

#pragma: issue a special command to the compiler or the linker. The example dictates the compiler to ignore a specific warning and the linker to link our code with OpenGL

```
#pragma warning( disable : 4705 )
#pragma comment( lib, "opengl32.lib" )
```

Preprocessor macros

The preprocessor has some built-in macros we can access in our code. They can be very useful when displaying error messages which can be enhanced with details about their location within our code.

__*LINE*__: an integer containing the current line of code

__*FILE*__: a string containing the current file name

__*DATE*__: a string containing the compilation date in the form 'MMM DD YYYY'

__*TIME*__ : a string containing the compilation time in the form 'hh:mm:ss'

__*cplusplus*: it is a long integer holding the version of the C++ standard the compiler supports. Possible values for Microsoft C++ compiler are

- std:c++14 (default) 201402L
- std:c++17 201703L
- std:c++latest 201704L
- Not specified 199711L

This macro is usually used to disable parts of code, mostly in headers, when compiling with C compiler. This is very common when we mix C and C++ code. The example code is a C++ header that can be included in a C file:

```
#ifdef __cplusplus
// the following definitions follow the C standard
extern "C" {
#endif
    // function that can be called both from C and C++ code
    void some_function();
#ifdef __cplusplus
// end of C mode
}
// C++ definitions can follow (always inside #ifdef/#endif block!)
class some_class{...};
#endif
```

Two classic examples

Here are two of the most common uses of the preprocessor.

We all use code to help us trace and fix errors when we build our projects. This code, however helpful it may be, slows the execution of the program significantly, and should be removed from our final built. Visual Studio generated projects define the macro *_DEBUG* for debug builds. We enclose our debug specific code in a block that the preprocessor will discard in the final built.

```
#ifdef _DEBUG
// DEBUG only code goes in here
#endif
```

Or we can leave code out of the compilation enabling or disabling features of the program by defining custom *macros*.

This code skipping trick with preprocessor macros has an even better use. The problem with big projects that have many source and header files is that some headers are *included* more than once due to cross reference. This leads to compilation errors because things are double defined. The trick we are going to use is really old. Modern compilers support the preprocessor directive *#pragma once* which solves the problem, but this might not be supported by all the compilers. If your code has to support a variety of compilers the old trick is the safest way to go.

```
/* high resolution timer */
// check if a custom definition exists
#ifndef __cg_timer__
// usually we define something close to the file name
#define __cg_timer__

// our definitions go here

#endif // __cg_timer__
```

Summary

In this chapter we covered the C++ preprocessor. A valuable tool that if used correctly can help us solve many problems. The advent of C++ has made many things we accomplished with the preprocessor obsolete. Actually, they are better done with C++. Afterall this is the reason behind C++, to do things better and write more robust code.