C++ tutorial

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 using *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 wrote code in *assembly language*, which is the actual list of direct commands to the processor. That kind of programming requires a lot of repetitive work and above all the programs 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.

C and C++ are compiled while Python is interpreted. This means that there is always an intermediate program that actually converts the code we develop into something the computer can execute. In the case of C/C++ it is a compiler, a program that converts our code into one piece of machine executable altogether, while in the case of Python it converts and executes one line at a time.

For the purposes of this tutorial, we will work on Windows 11, using Visual Studio 2022 Community Edition.

C++ is a 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 executable instructions*. Then we use another program called *linker*. The linker takes the output files generated by the compiler, checks 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 tutorial 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 tutorial.

You can download the sample code for this tutorial from https://github.com/cosfer65/cpp_tutorial

Part 1: Basic concepts

The objective of this tutorial is to see the base elements of the language. We will not cover all the details of the language, but we will just scratch its surface. Our goal is to get you motivated and start developing your first programs in C++. If you are really interested in the language, and it fits your needs, then you will definitely find a lot of resources, either on the internet or in good bookstores, that will help you understand and utilize the language to its fullest potential.

The samples we will create will run on any operating system, since we are only working with the language features, and not any specific operating system features.

In this first part we are going to see some fundamental concepts that will show you the very basics of the language

Our first program

We are starting with the classic program every book or tutorial starts for as long as I can remember. A program that greets the user, or as it is known the *hello world* application.

```
/* this is our first program
    created: 2024
    purpose: show basic concepts of C++
    copyright: add your name here
*/
#include <iostream>

// our main function, the entry point to our program
int main()
{
    std::cout << "Hello from part 1!\n";
    return 0;
```

Let us go through the code we see. In the top of the code, we see a part that is marked in different color and looks like plain English. It starts with */** and ends with **/*. This block is a *comment*. The compiler ignores it completely. So, we can write anything we want. We usually write comments to improve the readability of our code, or to keep track of the history of the file, as is the case with this comment. Further down the code we see another line with the same color that starts with a double slash *//*. This is another comment. In C++ we can start a comment with double slash and the compiler ignores whatever we write from that point until the end of the line.

The next is the line that starts win the directive *#inlcude*. This instructs the compiler to read another file. In this case it reads the file called *iostream* and it is part of the *Standard Library* of C++. It contains information on how to print information to the screen that we will need in our program. For the time being please accept it without much detail, it will become clear as we go along.

After the one line comment we see something strange, at least for a beginner. In C++, the code is organized in functions. This function is called *main,* and it is the most important function in C++ programming. According to the language specification the execution of our program starts from this function. Every program MUST have a *main* function, otherwise we get an error message that the *main is not found*.

A function has two main components. Its *header* and its *body*. The header is the first line *int main()*, and its body is enclosed in curly brackets. In the first line of the code, we greet the user by printing a simple message on the screen, and then we just return control to the operating system.

Keywords

C++ has a set of reserved keywords that have special meanings to the compiler and cannot be used for naming variables, functions, or other identifiers. Here are some of the commonly used C++ keywords:

- auto: Deduces the type of variable automatically.

- bool: Represents a Boolean value (true or false).
- break: Exits from a loop or switch statement.
- case: Defines a branch in a switch statement.
- catch: Catches an exception.
- char: Represents a character data type.
- class: Declares a class.
- const: Declares a constant value.
- continue: Skips the current iteration of a loop and proceeds to the next iteration.
- default: Specifies the default branch in a switch statement.
- delete: Deallocates memory that was previously allocated by new.
- do: Starts a do-while loop.
- double: Represents a double-precision floating-point number.
- else: Specifies an alternative branch in an if statement.
- enum: Declares an enumeration.
- extern: Declares a variable or function that is defined in another file or scope.
- float: Represents a single-precision floating-point number.
- for: Starts a for loop.
- goto: Transfers control to a labeled statement.
- if: Starts a conditional statement.
- inline: Suggests to the compiler to insert the function’s body where the function call is made.
- int: Represents an integer data type.
- long: Represents a long integer data type.
- namespace: Declares a scope for identifiers to avoid name conflicts.
- new: Allocates memory dynamically.
- private: Specifies private access control for class members.
- protected: Specifies protected access control for class members.
- public: Specifies public access control for class members.
- return: Exits from a function and optionally returns a value.
- short: Represents a short integer data type.
- signed: Specifies a signed integer data type.
- sizeof: Returns the size of a data type or object.
- static: Specifies that a variable or function has static duration or linkage.
- struct: Declares a structure.
- switch: Starts a switch statement.
- template: Declares a template.
- this: Refers to the current object.
- throw: Throws an exception.
- true: Represents the Boolean value true.
- try: Starts a try block for exception handling.
- typedef: Creates an alias for a data type.
- union: Declares a union.
- unsigned: Specifies an unsigned integer data type.
- virtual: Declares a virtual function.
- void: Specifies that a function does not return a value.
- while: Starts a while loop.

All these will be explained in this tutorial.

Variables

Programs are used to handle data and perform operations on them. So, every programming language has to provide us with a mechanism to handle and manipulate data. This is done with *variables*. Variables are primarily used to store data and then allow us to access and manipulate them.

C++ is a *strongly typed language*. This means that we must specify variable types so that the compiler knows how much space is required to store them, and more importantly how to handle operations involving them.

C++ has predefined built-in data types. The size of these types depends on the underlying operating system and architecture. First we will introduce the data types and then we will see the space they take up and the values they can handle.

1. Integers: *int* and *long*.
2. Floating point numbers: *float* and *double*.
3. Characters: *char*.
4. Booleans: *bool*.

The system architectures, also referred to as data models, which determine C++ data sizes are:

1. LP32 or 2/4/4: *int* is 2 bytes long, *long* and pointers are 4 bytes long. This is the Win16 API.
2. ILP32 or 4/4/4: *int, long* and *pointers* are 4 bytes long. This is the Win32 and Unix like 32-bit OSes.
3. LLP64 or 4/4/8: *int* and *long* are 4 bytes long, and *pointers* are 8 bytes long. This is the 64bit Win32 API.
4. LP64 or 4/8/8: *int* is 4 bytes long, *long* and *pointers* are 8 bytes long. Unix and Unix like systems.

The size of *long* in cases 3 and 4 is a fundamental difference between Windows and Linux, which are the most common operating systems. We must be careful if we plan to port our code to both operating systems. Here is the length of each data type in bytes.

Type specifier	C++ standard	LP32	ILP32	LLP64	LP64
char	1	1	1	1	1
bool	1	1	1	1	1
int	At least 2	2	4	4	4
long	At least 4	4	4	4	8
float	4	4	4	4	4
double	8	8	8	8	8

All data types except *bool* can have negative and positive values. Bool on the other hand can only be either *true* or *false*. We can modify the signed value behavior for *int* and *long* data types and use them to store *unsigned* values using the *unsigned modifier*. We can modify the storage requirements of *int* and *long* using the *short* and *long* modifiers.

Here is the list of the modified data types and their respective range:

data type	size in bytes	Range
int	4	-2147483648 to 2147483647
unsigned int	4	0 to 4,294,967,295
short	2	-32768 to 32767
Unsigned short	2	0 to 65535
long	4 or 8	-2147483648 to 2147483647 *if 4 bytes* -(2^63) to (2^63)-1 *if 8 bytes*
unsigned long	4 or 8	0 to 4294967295 *if 4 bytes* 0 to 18,446,744,073,709,551,615 *if 8 bytes*
long long	8	-(2^63) to (2^63)-1

unsigned long long	8	0 to 18,446,744,073,709,551,615
float	4	1.2E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
char	1	-128 to 127
unsigned char	1	0 to 255

Here are some example variables:

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

Constants

C++ introduced the concept of *constant.* These are values that remain fixed throughout the execution of the program. They are declared using the *const* keyword. They are assigned their values when they are initialized.

```
const int c = 10;      // const variable
const int& cr = c;     // reference to const variable
const int* cp = &c;    // pointer to const variable
```

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

Operators

Operating on our data is fundamental to every programming language. Operators are the special symbols we use to perform operations on our variables and data. These operations are usually mathematical or logical.

The operators operate on operands. A simple example we all learned early in our school years is addition:

```
int a, b, c;   // first declare some variables
a = 10;        // assign a value to a
b = a + 10;    // b is a plus a constant value
c = a + b;     // c is the sum of a and b
```

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 *ternary* operator, *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

Arithmetic operators are used to perform classic arithmetic operations like addition. The only addition of the language are the *increment/decrement* operators that are used to increment or decrement integer values.

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

Relational operators are used to compare variables. Nothing new was added to them by the language. They perform the comparison, and they return *true* or *false*.

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

Logical operators perform as we have learned in Boolean algebra. Anything different than zero (0) is considered to be true and anything zero is false. The Boolean combination of such values produces the respective result.

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

These operators operate on the variables on bit level. This is so easy to explain right now since we have to explain how data is stored in binary format and then how these operators work. We will address this later.

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

The first operator clearly assigns a value to a variable. The rest perform an operation on the variable on the left, using whatever is on the right, and then they assign the result to the variable on the left.

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
(&) address of	&a	address of operator (&) returns the address of variable
(*) dereference	*a	dereference operator(*) addresses the variable

Apart from the *conditional* operator which selects a result based on a logical condition, the rest of them ae used to convert variables, separate list items, or retrieve information about a variable.

Operator precedence

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. Mathematical operators in particular follow the precedence defined in mathematics, because otherwise we would end up with a language that could not be used for any scientific work.

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

In C++, the *statements* are the building blocks of a program. They are executed sequentially and can be categorized into several types:

Expression statements: These include expressions followed by semicolons, like this:

```
a = 10;
b = a + 10;
```

Compound statements: These are groups of statements enclosed in curly braces *{}*. Here is an example:

```
{
    int a, b;
    a = 10;
    b = a + 10;
}
```

Declaration statements: These are the statements where we declare our variables:

```
int a, b;
```

All but the compound statements MUST be terminated with a semicolon. This is the way the compiler understands the end of a statement. This convention allows us to write many statements in one line of code:

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

Although this is a perfectly valid piece of code, it has proven to be a bad practice, because putting so much stuff together makes the code hard for humans to understand, and eventually upgrade or fix.

Functions

A function is a block of code that performs a specific task. It can be called multiple times within a program, which helps in reducing code redundancy and improving readability. Using functions, we can break down a big and complex task into smaller steps easier to implement and maintain.

We can pass them data, referred to as *parameters* or *arguments*. They are used to perform certain operations such as counting the words in a sentence or calculating the square root of a number.

As we said in the beginning, C++ code must be inside 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 is the main function, and it is the *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)
{
    statement(s)
}
```

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

The function header tells us all 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 whenever 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++.

```
void simple_function() {
    std::cout << "this is a simple function\n";
}
```

This is a simple function that just displays a message on the screen and does not return any value, thus it is declared as *void*.

Header files

Header files in C++ are essential for organizing and managing code, especially in larger projects. They typically contain declarations of functions, variables, classes, and other entities that can be shared across multiple source files.

What are header files

These files allow us to declare the interfaces of your modules or libraries. This means we can define functions, classes, and variables in one place and then include them wherever needed.

Header files usually have the extension *(.h)* or *(.hpp)*.

By breaking our projects into multiple we make them modular and easier to maintain, while we make our code reusable. Header files are key files in this way of code organization.

Typically, we have two main categories of header files. These are the *Standard* headers versus the *User-defined* headers. The first are usually located where the *Standard Library* is and the compiler knows where to look for them, while the latter are located with our code, and we instruct the compiler where to look for them. Here is an example of header usage:

```
#include <iostream>    // Standard Library header
#include "part1.h"     // custom user-defined header
```

Namespaces

Large programs like a CAD application or a physics simulation 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 allow us to group together related entities and organize 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 similar tasks 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 {
    // declare some function
    return_type func_name(arg_type arg);
    // continue with other declarations
    other_declarations
}
```

And in the source file we write:

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

Accessing an entity within a namespace requires us to precede the entity name with the name of the *namespace* and the *scope operator '::'*.

Here we define our *sample_namespace* namespace and within this namespace a function that calculates the average of two numbers.

```
// the declaration in the header
namespace sample_namespace
{
    float calculate_avarage(float a, float b);
}

// the definition in the source
namespace sample_namespace
{
    float calculate_avarage(float a, float b)
    {
        float sum = a + b;
        float avg = sum / 2;
        return avg;
    }
}

// the function call
```

We can avoid using the namespace name when calling the function if we use the `using` keyword:

```
using namespace sample_namespace;
float fv = calculate_avarage(2.3, 4.5);
```

Although it is very convenient I would not recommend using it, especially in large projects. It is very convenient to know where the function you are using comes from. Functions in the *std* namespace for example are known to be stable and you should elsewhere should a problem ever emerges.

Basic I/O

One of the basic things a program needs is communication with the user. Either in the form of input when it gets input from the user, or as output when presenting results or execution options, or anything the programmer could ever imagine or need. A simple keyboard input or a message printed on the screen are the most basic operations a program should perform.

C++ has no built-in mechanism or *commands* we can use to perform any of these tasks. The designer of the language took this characteristic from the C programming language. Instead of having a built-in mechanism in the language it completely relies on libraries, usually written in the language itself, to provide this functionality.

The library in C++ is called *Standard Library*. It is a really big library because over the years it has adopted a lot of functionality covering many areas of programming. It is under the control of the International C++ Standardization Committee so we can be sure that it contains essential functionality and stable code we can certainly rely on in our projects. All the entities of this library are in the *std* namespace.

This library will give us the input and output functions we are going to use throughout this tutorial. After all the only things we need are reading some user options from the keyboard and displaying some messages.

To use the input/output functionality of the library we need to *include* the *<iostream>* header. This contains all the required information for the compiler about the I/O functionality. The *Standard Library* is broken down into many modules, based on the functionality they provide. This improves compilation time because otherwise it takes too long to read all definitions of the library.

In the library the input and output mechanisms are treated as streams of information, so the name *iostream* was given the header. From this functionality we will need only two *streams*, the *cin* for input, and the *cout* for output.

In our example we prompt the user to input a number, then we read it and finally we display it:

```
#include <iostream>
void basic_io()
{
    double a;
    std::cout << "insert a:";     // give the user a hint
    std::cin >> a;                // read the value of a from the keyboard
    std::cout << "input was:" << a << "\n";  // print what the user typed
}
```

Invoking the I/O functionality looks a little strange. We are not using a typical function call syntax. Instead, we are using these odd-looking *operators*. These are the streaming operators, pushing data to the output stream, or pulling from the input stream.

The direction of the operator is actually showing the *flow* of data. In the case of output, we have the line

```
std::cout << "insert a:";
```

Here we have the output stream, *std::cout*, and we are pushing the string *"insert a:"* to the stream to be displayed on the user's screen.

In the next line

```
std::cin >> a;
```

we wait until the user has typed a number and the input stream, *std::cin*, has copied it to our variable.

Finally, we create a multiple output by sequentially streaming a prompt,

```
std::cout << "input was:" << a << "\n";
```

our variable and the *new line* character.

Escape characters

Since we have presented it several times so far when we print something on the screen, it should be fair to explain it. At the end of our output, we issued the *new line* character. This strange two-character string instructs the compiler to send the *new line* character, a special character in computer systems that actually moves the cursor to the beginning of the next line. We must point out here that although we write two characters the *std::cout* stream translates it to the *new line* character.

This is not the only character sequence that gets converted into something else before it is sent to the system. This group of character sequences are called *Escape Characters,* and they are used to control text output. Initially they were designed for printers in the first days of computing when printer type devices were used instead of monitors. Back then it was essential to make a beep, move backwards or move to the next line.

Here is the list of the *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

Summary

- C++ is a compiled language
- We got acquainted to the basic keywords of the language
- We have seen the basic syntax of C++
- C++ is noticeably light
- Everything comes in the form of external libraries
- Code is organized in functions which can take arguments and return values

Part 2: Controlling the flow of our code

Dealing with data means we must do a lot of operations on them. This means we have to repeat some process, take different paths, and alter the flow of the code to achieve our goal. There are endless examples of everyday life where we go through data to solve some trivial problem.

Imagine the process of calculating your average score in the exams, or finding the longest of some sticks, or even harder finding the path from your home to your job.

The logical process to solve a problem is called *algorithm*. Algorithms are complex processes and can only be completed if we have the ability to modify the flow of our though. Straightforward approach is only useful when we add two numbers. The harder the problem the more flexible our thought process must be. The same rule applies to computer programs as well.

Branching, or taking a different path

We are starting with *branching* our code to different paths. The first example that comes to mind is how we could notify a user whether he passed the exams or not.

If - else

In our mind we look at his grade and if it is greater than or equal to a certain value we tell he passed, otherwise we tell him he failed. This is actually the first algorithm we create to solve a problem. Here is he code for this sample:

```
#include <iostream>

int main() {
    double a;                                       // declare our variables
    std::cout << "tell us your grade (1-20):";    // give the user a hint
    std::cin >> a;                // read the value of a from the keyboard

    if (a >= 10) {
        std::cout << "pass\n";
    }
    else {
        std::cout << "fail\n";
    }
    std::cout << "thank you\n";
```

Now we will go through the code and analyze the syntax of this statement. As expected we have the *if* keyword followed by a comparison operation in the parenthesis. Then we have a statement in curly brackets, the *else* keyword and finally another statement in curly brackets. Let us analyze these five blocks one by one.

The first (*if*) is the basic keyword to signal that we want to perform a branching statement. As in real life it is followed by the rule we have to check. In programming terms this is a *conditional*. It is an expression that results either in *true* or *false*. For a computer anything that equals to zero (0) is false, and everything else is true. After the conditional comes the *compound statement* that has the code to be executed if the conditional is true. In the general case we can have another path to follow in case the conditional is false. That path is the *compound statement* that follows the *else* statement. When everything is done the code resumes at the statement following the last compound statement.

Here is the official syntax of the *if* statement:

```
if (condition is true)
    compound statement
else
    compound statement
statement(s)
```

```
if (condition is true)
    compound statement
statement(s)
```

Conditions

Before we continue with any other branching method the language offers, this is a good opportunity to see these *conditionals* from a little closer and understand them because they are used in many cases in C++ and in programming in general.

As we mentioned before a *conditional* is any expression that can be evaluated as either *true* or *false*. The most profound expressions are those where we check for equality between values. In our example we check whether the user's mark is greater or equal to 10. These equality check operators are called *relational operators,* and we saw them in *part1*. Here they are:

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

C++ has introduced a new type of data called *bool*. This can hold only two values: *true* and *false*.

Apart from real Boolean values though our conditionals can have any variable or function call. Traditionally any non-zero value is considered as true, and a zero value is false. Here is an example:

```
// a function returning always 0
int zero_return() {
    return 0;
}
// and a function returning always true
bool is_valid() {
    return true;
}
```

The code calling these functions would be like:

```
// since the function returns 0, this is always false
if (zero_return()) {
    std::cout << "if you see this message something went wrong\n";
}
// this is always true
if (is_valid()) {
    std::cout << "this is a really valid result\n";
}
```

We can combine conditions using the *logical operators* we saw in part 1:

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

The combined logical result follows the rules of *Boolean Algebra*. The order of precedence for the logical operators is from left to right. Depending on the logical operator it is possible that the compiler will generate code that does not execute the second conditional if the first is sufficient. Look at these examples:

```
// for AND both must be true
// this executes both if the first is true
// and only the first if the first fails
if (value > 10 && value < 20) {}

// for OR only one needs to be true
// this may execute only the first and if it fails
// it executes the second
if (value > 10 || value < 20) {}
```

You should keep this in mind and never rely on conditional to execute any code but only perform logical checks.

switch - case

The *switch* statement in C++ is a control statement that allows you to execute different blocks of code based on the value of an expression. It is often used as an alternative to a series of *if … else if* statements, making the code cleaner and easier to read.

Here is the syntax for this statement:

```
switch (expression) {
case constant1:
    // code to be executed if expression equals constant1
    break;
case constant2:
    // code to be executed if expression equals constant2
    break;
    // you can have any number of case statements
default:
    // code to be executed if expression does not match any case
    break;
}
```

So, this code:

```
// user menu option processing
if (menu_selection == 1) {
    std::cout << "one\n";
    // execute respective code
}
else if (menu_selection == 2) {
    std::cout << "two\n";
    // execute respective code
}
else if (menu_selection == 3) {
    std::cout << "three\n";
    // execute respective code
}
else {
    std::cout << "selection out of bounds\n";
    // execute respective code
}
```

Can be rewritten like this:

```
    // user menu option processing
    switch (menu_selection) {
    case 1:
        std::cout << "one\n";
        // execute respective code
        break;
    case 2:
        std::cout << "two\n";
        // execute respective code
        break;
    case 3:
        std::cout << "three\n";
        // execute respective code
        break;
    default:
        std::cout << "selection out of bounds\n";
        // execute respective code
        break;
    }
```

It is easier to follow this code because we know that all the *cases* are testing the save expression against the given values, where in the first implementation we need to check them one by one.

With this command we introduce the *break* statement. When encountered, this statement transfers control to the first statement after the *compound statement,* it appears, in this case after the switch. If we do not issue the break in some *case*, the code will continue executing the next *case*. Try commenting out one *break* and see the results. This feature in the design of the command gives us some flexibility when designing code, but it comes with the cost of extra care we have to put when designing and writing our code.

Loops, performing repetitive tasks

Many times, in our programs we have to do the same thing repeatedly. Rewriting the same code though is not very productive and in general it is error prone. It is not practical either limiting us to a small number of options.

For this reason, programming languages incorporate *loops* in their syntax. Loops are especially useful for several reasons.

- *Repetition*: They allow us to execute a block of code multiple times without having to write the same code repeatedly. For example, if we want to print “Hello World” 100 times, we can use a loop instead of writing 100 print statements.
- *Efficiency*: Loops make our code more efficient and easier to manage. They reduce redundancy and the potential for errors that come with copying and pasting code.
- *Flexibility*: Loops can handle varying numbers of iterations based on conditions. For instance, we can use a loop to process each element in an array, regardless of the array’s size.
- *Flow Control*: They provide a way to control the flow of our program. We can use different types of loops (for, while, do-while) depending on the specific needs of your task.
- *Complex Operations*: Loops enable us to perform complex operations, such as calculating the sum of a series of numbers, iterating through data structures, or implementing algorithms.

while

The *while* loop in C++ executes a block of code repeatedly as long as a *logical* condition is *true*. Here is the typical syntax of the loop:

```
while (condition) {
    // code block to be executed
}
```

The code within the loop block starts executing if the *condition is true* and stops executing when the *condition is false*. It is important to remember that the code in the loop might *never* run if the condition is *false*.

In practice we have some code before the loop to do some initialization and some code in the loop that might change the value of the *condition*. Here is the updated version of the loop:

```
initialize_condition;
while (condition)
{
    statement(s);
    step_condition;
}
```

Here is how we can use the while loop to calculate the sun of the numbers from 1 to 5:

```
// calculate the sum of 1->5 without a loop
// this is a fixed calculation
int sum = 1 + 2 + 3 + 4 + 5;

// do the same with the while loop
sum = 0;  // reset the sum
int i = 1; // initialize the condition
// while i < 6(not included) means from 1 to 5
while (i < 6) {   // check the condition
    sum += i;
    i += 1;    // move to the next number
}
```

The first calculation is a fixed addition of five numbers. If we look at the loop we can see that we can make it more flexible. We can modify the conditional and instead of going up to 5 we can use a variable and go as far as we want. Likewise, we can modify the starting value as well as the step ending up in a function that can actually calculate any sum we want:

```
int calc_sum(int start, int finish, int step) {
    // do the same with the while loop
    int sum = 0;  // initialize the sum
    int i = start;    // initialize the condition
    // while i < finish(included) means from 1 to 5
    while (i <= finish) {   // check the condition
        sum += i;
        i += step;    // move to the next number
    }
    return sum;
}
```

The fixed summation of the values just does not work.

We can make the loop run forever if we put in the conditional the value 1. Endless loops are used in programs we do not know how many times we will have to repeat the loop. For example, wait until the user chooses to terminate the program, and in any other case do some processing.

for

The while loop is not our only option in C++. A similar loop with *while* is the so called *for* loop. We usually visit the *for* loop after the *while* loop because it adds some things to its syntax, but otherwise it is almost the same. We take a look at its syntax and then we will break it down:

```
for (initialization_statement; condition; step_statement)
    compound_statement
```

As we can see it is the same as the *updated* version of the *while* loop we developed. The two loops differ only in syntax. Otherwise, they are completely the same. Especially if we keep in mind that we can omit the `initialization_statement` and the `step_statement`. In that case the loop will be:

```
for (; condition;)
    compound_statement
```

Notice that we have left the semicolons. We can leave out the *conditional* as well, NOT the semicolons, and the loop will run forever.

The generated code for this loop starts with the *initialization_statement*. Then it checks for the *condition*. If it is *false* the loop will not run. At the end it executes the *step_statement*. After that it goes back to check the *condition,* and it is *true* it runs the loop or if it is *false* it continues with the first statement that comes after the loop.

Like the *while* loop, this loop also checks for the *condition* at the beginning.

do … while

Finally, we have one more loop. This is the *do…while* loop. Here is its syntax:

```
do
    compound_statement
while (condition);
```

Our sum calculation function using this loop will become:

```
int calc_sum_with_do(int start, int finish, int step) {
    int sum = 0;  // initialize the sum

    int i = start;
    do {
        sum += i;
        i += step;
    } while (i <= finish);
    return sum;
}
```

There is a significant difference between this and the previous two loops. In this statement the check for the condition is done at the end of the loop. This means that the code in the loop will run at least once. We may either have a variable we initialize before the loop to control it, but the most common case is terminating the loop by checking one or more variables that are set within the loop:

```
void wait_for_input() {
    int value;  // just declare the variable
    do {
        std::cout << "input a number between 1 and 10:";
        std::cin >> value;
    } while (value < 1 || value >20);  // as long as we are out of bounds, repeat
    std::cout << "value:" << value << "\n";
}
```

Getting out of the loop

To add flexibility in loops C++, much like any language that respects itself, gives us the ability to modify the flow of the code in a loop. This can be achieved with the use of two statements: *break* and *continue*. Here is a more detailed presentation of the two.

break

This statement terminates the loop immediately and transfers control to the first statement after the loop. Needless to say, it skips any remaining code that comes after it within the loop. It usually comes within a branching statement. It is clear that issuing a break command without a check if it is appropriate is not a particularly good practice.

Being able to terminate the loop should something unexpected happen, gives us the flexibility to add one more condition to check while iterating.

We will use the *while* loop to present the syntax of the command, and for the example code. Once we understand its behavior and syntax, it is obvious that it is exactly the same no matter what looping statement we use.

```
while ( condition )
{
    statement(s)
    if (we_should_break)
    {
        break;
    }
    [optional statement(s)]
}
code_comes_here_after_break
```

As we said, although we have the main *condition* to control the loop we can also have one more control point where we check and terminate the loop, skipping some code.

We will modify our *calc_sum* function to take an extra parameter, the maximum allowed value for *sum*. Then our function would be like:

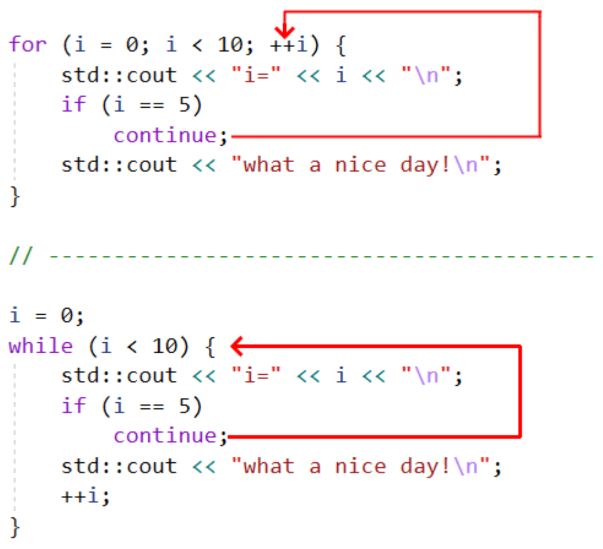
```
int calc_sum(int start, int finish, int step, int max_limit) {
    int sum = 0;  // initialize the sum
    int i = start;    // initialize the condition
    while (i <= finish) {   // check the condition
        sum += i;
        if (sum > max_limit) { // limit was passed?
            sum -= i;   // need to go back
            break;   // and terminate the loop
        }
        i += step;    // move to the next number
    }
    return sum;
}
```

continue

There are cases though, where we do not want to break the loop but rather go back to the beginning, skipping some of the code. In that case we can use the *continue* statement. This statement does just that.

```
while ( condition )
{
    statement(s)
    if (we_should_skip)
    {
        [optional statement(s)]
        continue;
    }
    [optional statement(s)]
}
```

Depending on the loop and the way we write our code, there might be some unexpected and nasty results. Take a look at this image and note where the control is transferred after *continue*.



```
for (i = 0; i < 10; ++i) {
    std::cout << "i=" << i << "\n";
    if (i == 5)
        continue;
    std::cout << "what a nice day!\n";
}

// ------------------------------------------

i = 0;
while (i < 10) {
    std::cout << "i=" << i << "\n";
    if (i == 5)
        continue;
    std::cout << "what a nice day!\n";
    ++i;
}
```

In the case of *for* loop, the control goes to the *step* statement, which allows the loop to continue its operation normally. Although the second loop is supposed to do the same thing, *continue* transfers control to the *conditional check*, the variable is *not* incremented, and the loop continues forever. The same would have happened if we had omitted the stepping statement in the *for* loop as well.

This means that we must be twice as careful when we use the *continue* statement, because it may lead to really bad situations, which may be extremely hard to find if our code is long and complex.

goto

This is one of the simplest commands in any language that supports it. It issues a *jump* instruction to the CPU transferring control to a certain point in the code, defined by the *label* following the command. The *label* could be before the *goto* command, resulting in some kind of loop, or after the *goto* and skip some code. In the early days of computing when languages did not have so advanced features, our programs were small, and we were struggling for execution speed, this command was really useful.

As our programs grow both in size and complexity, using *goto* may lead to code that is hard to understand and of course to maintain. We have to follow the code, find the *label*, and try to guess what it is all about.

We must keep in mind that the most expensive part of software development is maintenance, not initial development.

The years have passed and now our languages have evolved, our programs have grown enormously, and the computers are fast enough to handle the most complicated tasks. The new features of programming languages have made features like *goto* almost obsolete. So far I have encountered only *one (1)* case where we could not implement an algorithm without the use of a *goto*.

Comparing a *for* loop in C++ with a piece of code that does the same thing using *goto* will make this statement clear. Here is the code using the *for* loop:

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

Perfectly clear for anyone reading it. Here is the same loop using *goto*:

```
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

continue_goes_here:  // continue jumps here (with a goto as well)
step;   // perform the step
goto start_of_loop;  // and go back to the start
end_of_loop: // break jumps here with goto
```

Technically this is the kind of code the compiler generates. But this code is *rebuilt* every time we change the C++ above, and not fixed or maintained, operations that may lead to errors.

return

This command is used in functions when we want to return to the caller. It can appear anywhere within the function, and it immediately returns control to the point the function was called. In the case of functions that have a return type other than *void* we also have to specify a value to return.

We usually find it at the end of a function returning some result after the function has manipulated the data, or nothing in case of *void* functions. Being able to place it anywhere within the function can simplify the function code significantly.

Consider a function that calculates the square root of a number. It expects a positive number, we leave imaginary numbers out of the picture. The algorithm to calculate the square root will fail with unexpected results. We have two options to make our code robust. One is to put as many checks as possible to avoid nasty behavior, or check the number passed and if it is negative return some predefined error value, and if it is positive process with simpler code since we operate on valid data. Here is some fiction code that implements *square root*:

```
double  calc_sqrt(double num) {
    if (num < 0)  // no square root for negative
        return 0; // stop processing and return
    double sq_rt = 0;
    // do the algorithm to calculate square root
    // store it in sq_rt, and return it
    return sq_rt;
}
```

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

Part 3: Functions

As we saw earlier, 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 part we will take an in-depth look at functions and learn those features that will help us create efficient and robust code.

Functions

In programming we break our code into functions. This practice has a lot of benefits. Here we are listing some of them.

1. Modularity: we can break complex problems into smaller tasks easier to handle, understand and maintain.
2. Reusability: these smaller functions can be used as part of a solution for another problem. This reduces redundancy and saves us a lot of time.
3. Readability: the code is now organized and easier to read and understand. We see functions with clear names describing what they do, instead of complex code.
4. Maintainability: breaking code int autonomous functions makes it easier to locate possible problems and fix them.
5. Testing: it is easier to test simple and small functions and make sure they do what we intended. This makes the whole process of debugging our software a lot easier.
6. Abstraction: none needs to know how a certain function works. They only need to know what it does and how to use it.
7. Collaboration: breaking the problem into smaller tasks means that more people can work on it without interfering with each other and develop the solution faster.

We have two main types of functions:

1. **Standard Library Functions**, which are predefined by the language
2. **User-defined Functions**, developed by the user

So far we have seen some Standard Library Functions like *std::in* or *std::cout*. We have also seen the user defined function main, which is required by the language, but we have to write. In this part of the tutorial, we will focus on User-defined Functions, and how we can organize them in our projects.

Function Definition

In C++ we create our own functions. Here is what a function looks like:

```
returnType functionName(argument_list) {
    // function body
}
```

Functions in C++ can return a result. The type of this result is the *returnType*. This can be an integer, a double and so on. If it does not return anything we say it is of type *void*. The *argument list* can be a list of variables or values, depending on the function, which will be passed to it in order to get the results we need.

Here is a simple example function:

```
int simple_function(int arg1, double arg2){
    // print the arguments received
    std::cout << "arg1=" << arg1 << ", arg2=" << arg2 << "\n";
    // and return some value
    return 3;
}
```

It does not do anything fancy but we can experiment with the arguments we pass to it and the values it returns.

Calling a function

Ow that we have created a function we need to call it. All we have to do is write its name and the arguments we want to pass, and the compiler will figure everything out for us:

```
// call the function and get the return value
int ret = simple_function(8, 2.34);
std::cout << "function returned:" << ret << "\n";

// we may ignore the return value
simple_function(8, 2.34);
```

As we see in the example we can call the function and ignore the return value. Be very careful when you ignore the return value of a function, because it is there for some reason, the simplest being a way to let you know if it succeeded or failed to do what it was supposed to do.

Declaring a function

As we said in the beginning, C++ is a strong typed language. Everything has to have a concrete type that cannot change throughout the program execution. The compiler has to know all the entities and their exact definition before using them. To assist with the declarations of the program entities we us the *header files*. We have seen them in **part 1**. So now we are going to put our function declarations in headers.

Declaring a function is not so hard. All we need is the *returnType*, the *functionName* and the *argument_list*. This is the first line we saw in the syntax of the function definition earlier on. Here is the syntax of the function declaration:

```
returnType functionName(argument_list);
```

And our *simple_function* would be declared like this:

```
int simple_function(int arg1, double arg2);
```

Function overloading

Function Overloading is a feature of the language that allows us to have multiple functions with the same name. For the compiler ,and the users of our code of course, these functions must have different arguments. This means that they should have:

1. **Different number of arguments**.
2. **Types of arguments**, the arguments of each function must be of different type, for example one *integer* and one *double* versus two *doubles*.
3. **Different order of arguments**, for example one *integer* and one *double* versus one *double* and one *integer*.

Upon compilation the compiler determines the correct function to call based on the types of the arguments we pass in the call. This is one of the most profound reasons why C++ needs a declaration of everything, either variables or functions. One thing we must mention here is that overloading is based *only* on the argument list and not on the return type of the function.

Here are the declarations of some possible versions of the *add_numbers* function:

```
// the declaration of a function that adds two numbers
double add_numbers(double a, double b);
// a function that adds three numbers
double add_numbers(double a, double b, double c);
// or add a double and an integer
double add_numbers(double a, int b);
// or add an integer and a double
double add_numbers(int a, double b);
```

This code is calling the functions, and the compiler will generate the correct call:

```cpp
int main() {
    // calculate and store the sum of two numbers
    // the function is declared in part4b.h and
    // implemented in part4b.cpp
    double result = add_numbers(2.3, 4.5);

    // three doubles
    result = add_numbers(2.3, 4.5, 6.7);
    // or a double and an integer
    result = add_numbers(2.3, 4);
    // or add an integer and a double
    result = add_numbers(5, 6.7);

    return 0;
}
```

An obvious benefit of this feature of the language is that we do not need different names for functions that actually do the same thing, making I easy to remember and reuse our code.

We can skip the declaration of a function as long as its definition comes before it is used. In that case we can say that the definition acts as declaration as well, but only for functions that are written after this function. This is not a good practice though because the function is not publicly available.

Default arguments

In C++, *default arguments* allow us to specify a value for a function parameter that will be used if no argument is provided for that parameter when the function is called. This can make our code more flexible and easier to read.

There are some rules we need to follow when we specify default arguments for a function:

1. Default values must be specified in the function declaration.
2. All subsequent parameters after a default parameter must also have default values.

Here is an example to illustrate *default arguments*:

```cpp
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 we said before this definition acts as a declaration too, so we can specify our default arguments. So, we should declare our function and define it like this:

```cpp
// our function declaration with the default arguments
int multiply(int i = 1, int times = 1);

// the definition cannot declare the default arguments
// we write them in comments so we do not forget
int multiply(int i /*= 1*/, int times /*= 1*/)
{
    return i * times;
}
```

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.

Using *default arguments* makes our code 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.

Functions & variables

Type is only one attribute of a variable. It 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 released when they go out of 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 arguments of the function are local variables as well. The only difference is that they are initialized automatically with the values we provide in the function call.

The following example shows it all:

```
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

    // i is a local variable so the next statement is perfectly valid,
    i = 100;

    if (i > 10) {
        double l = 100 * d;
    }

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

Static local variables

Static local variables are variables that retain their values between function calls. Regular local variables are created and destroyed every time a function is called. Static local variables on the other hand, are initialized only once and persist for the lifetime of the program.

Static local variables are visible only the function, just like ordinary local variables. This means they do not mess with the global namespace and cannot be used to transfer data between functions. Yet they give us the persistence of a global variable.

In the example we create a simple counter to count how many times a function is called. The counter is created the first time the function is called. This means the line `static int localCount = 0;` is called only once. For this reason, it is essential to add the initialization in this line as well. In any subsequent call this line is skipped.

```
int static_variables() {
    // count how many times the function was called
    static int localCount = 0;
    ++localCount;
    return localCount;
}
```

Break the program in multiple source files

Breaking your code into multiple files can offer several benefits:

Improved Readability: As your codebase grows, keeping everything in a single file can make it difficult to navigate and understand. Splitting code into smaller, focused files makes it easier to read and maintain.

Better Organization: By separating code based on functionality or concerns, you can keep related pieces of code together. For example, you might have one file for user interactions and another for data processing.

Easier Reusability: When code is modularized into separate files, you can easily reuse components across different parts of your project or even in other projects.

Simplified Collaboration: In a team setting, having code split into multiple files allows different team members to work on different parts of the project simultaneously without causing conflicts.

Enhanced Maintainability: Smaller files are easier to test, debug, and update. If a bug arises, you can quickly locate and fix it without sifting through a large monolithic file.

Faster Compilation: In languages like C++, splitting code into multiple files can speed up the compilation process since only the modified files need to be recompiled.

Here are some basic steps we take towards that goal:

Identify Modules: Determine the different modules or components of your project. For example, if we're writing a game, we might have modules for graphics, input handling, game logic, etc.

Create Header Files (.h): For each module, we create a header file that declares the functions, macros, and data structures that will be used by other parts of the program. For example, *graphics.h* might declare functions like *initGraphics()* and *drawSprite()*.

Create Source Files (.cpp): Implement the functions declared in the header files in corresponding source files. For example, *graphics.cpp* would contain the definitions of *initGraphics()* and *drawSprite()*.

Use Include Guards: Ensure that each header file has include guards to prevent multiple inclusions. This is typically done using #ifndef, #define, and #endif preprocessor directives.

Static functions

In C++ all functions are globally accessible. We just declare them in the header, and they are ready to use in our project. Breaking our project into files gives us the opportunity to hide the internal details of our code. The *initGraphics* function for example can be broken into simpler tasks like *initGPU* and *setGraphicsMode*. These two functions can also reside in the *graphics.cpp* file and be completely inaccessible from anywhere in our code, but the code inside the *graphics.cpp* file.

This can be done by declaring them *static*. It is better to declare them at the top of the file and then define them anywhere within the file.

Here is *graphics.h*:

```
#ifndef __graphics_h__
#define __graphics_h__

int initGraphics();
int drawSprite();

#endif //__graphics_h__
```

And here is *graphics.cpp*:

```
#include <iostream>
#include "graphics.h"

static int initGPU();
static int setGraphicsMode();

int initGPU() {
    return 1;
}

int setGraphicsMode() {
    return 1;
}

int initGraphics() {
    initGPU();
    setGraphicsMode();

    return 1;
}

int drawSprite() {
    return 1;
```

Global variables

Global variables in C++ are variables that are declared outside of any function, typically at the top of the source file.. They can be accessed and modified by any function or class within the same program. We can access them from other files as well if we declare them.

To declare a global variable outside the file it was defined we need the *extern* keyword. This declares the variable's name and type to the compiler but prevents it from allocating space in computer memory for it. The memory will be allocated be the code generated from the definition.

The following example will make it all clear. In the *add_functions.cpp* file we define a global variable named *myGlobalCount* like this:

```
// a global variable that counts
// how many times our functions are called
int myGlobalCount = 0;
```

In the header file *add_functions.h* we declare the variable like this:

```
// declare a global variable as extern
extern int myGlobalCount;
```

The definition of the variable was put in the beginning of the *add_functions.cpp* file so it is completely accessible from any function within that file. On the other hand, declaring it in the *add_functions.h* makes it accessible to every function in *functions.cpp* file that has *included* the *add_functions.h* header.

Global variables are very convenient, they are accessible from anywhere in the program, let us easily share information between functions and they are persistent throughout the execution time of the program.

But there is a price we have got to pay for this. First of all, they *clutter* the global namespace increasing the risk of name conflicts. Allowing everybody to modify them can lead to errors that are very hard to find. Finally, they reduce program modularity.

So, we need to balance the pros and cons very carefully when we define global variables.

Static global variables

Global variables can also be declared with the *static* keyword. This limits their *visibility* only in the file they are declared, and they are inaccessible from other source files. They *cannot* be declared as *extern* in a header and used in other files, because that would lead to a *link error*, the *linker* would not find the appropriate declaration when assembling (linking) the code to one executable.

Function pointers (pointers to functions)

Function pointers are pointers that store the addresses of functions. They provide an alternative way to call a function, or dynamically pass the function to call as an argument to another function. We can obtain a pointer to a function by using its name. Here is an example of using a *function pointer*:

```cpp
#include <iostream>

void function(int i) {
    std::cout << "i=" << i << "\n";
}

// take the function to use as argument
void check_out(int value, void (*fun_ptr)(int)) {
    // use the function we were told
    fun_ptr(value);
}

int main() {
    // get the function pointer
    void (*fun_ptr)(int) = function;
    // and use it to call it
    fun_ptr(33);

    for (int i = 0; i < 4; i++) {
        // pass the pointer as an argument
        check_out(i * 3, function);
    }
}
```

In the beginning we have the function we will call using a pointer. To declare a variable that is a pointer to a function we use the syntax: *return_type (*var_name)(arg_list)*. The name of the variable must be in the parenthesis with the address operator because otherwise the compiler assumes we mean a *return_type** due to operator precedence. Using function pointers is a very straightforward operation and helps us write flexible an dynamic code.

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

Part 4: Pointers & References

Pointers are a very powerful tool we have in C++. They allow us to access the memory directly to read or alter its contents.

Pointers

A *pointer* is a new type of variable. It is used to store the *address* aka the position in computer memory of another variable. This allows you to reference and manipulate the data stored at that address.

Here are some common uses of pointers:

Dynamic Memory Allocation: Allocate memory during runtime using *new* and *delete*.

Function Arguments: Pass large structures or arrays efficiently.

Data Structures: Implement complex data structures like linked lists, trees, and graphs.

Besides the obvious advantages we have from the common uses, pointers is a very powerful characteristic of C++ for two more reasons

Efficiency: using pointers we can have direct access to the memory which can make our programs faster.

Flexibility: it is easier to use and manipulate complex data structures required by big programs.

But as usually there is a price we got to pay for all this:

Dangling Pointers: Pointers that reference deallocated memory.

Memory Leaks: Forgetting to free dynamically allocated memory.

Dangerous if not mastered: Can lead to errors if not handled carefully.

Pointer syntax

To declare a pointer, we use the *asterisk (*)* symbol, also known as the *dereference operator*:

```
int* intPtr;   // a pointer to an integer
```

A pointer needs to *point* to a variable. For this we use the *address off operator (&)*.

From now on we have two ways to access the contents of *int_var*. One via the variable name like we have done so far and one via the pointer:

```
int* intPtr;        // a pointer to an integer
int int_var;        // an integer variable
intPtr = &int_var;  // the pointer now points to the variable
```

Note that the variable *intPtr* is an address in the computer memory so to access its contents, aka the value of *int_var*, either to modify or to read, we *have* to use the *dereference operator (*)*.

```
int_var = 10;  // setting the value
std::cout << int_var << "\n";
*intPtr = 12;  // and changing it
std::cout << int_var << ", " << *intPtr << "\n";
```

Dynamic Memory Allocation

In our programs we need to handle varying amounts of data. This raises the need for dynamic memory management. We need the flexibility to reserve different sizes of memory based on our needs. The operating system has these mechanisms and in C++ we can access them with *pointers*.

A pointer can point to a chunk of memory the system has reserved for us. Then se can use it to store and retrieve data for our needs.

The new operator

We use the *new* operator to request memory from the system:

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

This allocates one item of type *data_type*. We can allocate as many as we need to. In the example we allocate an arbitrary number of integers:

```
int many = 100;
int* dynPtr = new int[many];
```

We use a variable in the allocation to show that the call to the *new* operator is dynamic. If the allocation succeeds, we end up with space enough to store 100 integers. Say we want to hold the scores of 100 players in a game.

Accessing the dynamically allocated memory

To access the allocated memory, we have two different ways. One is using *pointer arithmetic* and one is using *array* notation.

Pointer Arithmetic: since a pointer is actually a variable containing a memory address if we modify its value it points to a different location. Note the difference between the pointer and the address it points to. Look at this code:

```
++dynPtr;
```

This increases the pointer by one (1) making it point to the next position in memory. It *does not* increase the value of the contents in the location it points to. To access the contents, we have to use the *dereference operator (*)*.

The nice thing about pointer arithmetic is that the increment operator advances the pointer to the next element based on the allocation type, in our example integer. Similarly, the decrement operator goes back one element. We can even add an integer, and the pointer will advance as many places as the value of the integer.

In this example we go over all the allocated memory and set some value:

```
int many = 100;
int* dynPtr = new int[many];
int* hiScores = dynPtr;  // another pointer to the same location
// iterating over the allocated memory with pointer arithmetic
for (int i = 0; i < many; ++i) {
    *hiScores = i; // set the value
    ++hiScores;    // advance the second pointer
    // adding an index to the first pointer
    std::cout << *(dynPtr + i) << "\n";
}
```

We created a second pointer and made it point to the same location as the first pointer. There is no limit in the number of pointers we can have pointing to the same location. We use both the pointer increment and the indexing to access the memory. Here we show the difference between modifying the pointer and modifying the contents of the memory it points to.

The big mistake programmers make when dealing with pointers is accessing memory beyond the limits of the memory allocated. If you change the loop limit from *many* to *many+10* and run the program, it will crash when the pointer goes path the last allocated entity.

The delete operator

When we no longer need the memory we must return it to the operating system. If we keep requesting for memory we do not return to the system we end up with what is called *memory leak*. This may lead even to system crash.

The *delete operator* is used to free the allocated memory and return it to the system:

```
delete [] dynPtr;
```

The *square brackets* tell the *delete operator* that we are not deallocating a single entity but an array of entities.

Here is how we treat one entity:

```
int* somePtr = new int;
delete somePtr;
```

The pointer we used to point to an integer variable *cannot* be deleted. Calling *delete* on it will lead to a compilation error.

A common mistake with pointers is continue using them after they have been deleted. In the example above we had two pointers referencing the same memory: *dynPtr* and *hiScores*. After deleting *dynPtr* we can still use them both. Playing around with the pointer itself is safe, after all it is just a variable, the problems start when we try to access the memory it points to.

The code in this example leads to a crash:

```
delete [] dynPtr;
hiScores = dynPtr;
for (int i = 0; i < 10; ++i) {
    *(hiScores + i) = 1000;
}
```

This code has the second of the two big mistakes which are common when dealing with pointers. It is attempting to directly modify the contents of the memory after it has been deleted. The first being the out of bounds memory access.

Create an array in C++

In C++ we can create an array that can be as big as we need, using the *standard library*. These arrays have a built-in mechanism that performs the *allocation* of the required memory and to *release* that memory when no longer needed. We still have the benefits of *dynamic memory allocation* without the low-level management. They rely on the characteristics of the language we will examine later.

Such a dynamic array is the *std::vector*. It is declared in the header *<vector>* and can contain any kind of data we like. We specify the data it will hold upon declaration of a vector type variable:

```
std::vector<int> vec{ 1,2,3,4,5,6,7,8,9,10 };
```

Here we create a vector of integers and initialize it with some data. When the variable goes *out of scope* the memory is released and we do not have to worry about any memory leaks.

Function arguments

Pointers give us a lot of flexibility when used as functions arguments.

Pointers as references to variables

Pointers are used in function arguments when we want to pass the value of a variable but a reference to it. This means that the called function knows the location of our variable and can read or modify it instead of getting just a copy of it. Passing a reference of our variable is essential when we need the called function to modify its value.

Here is an example of this:

```
int byrefInt(int* val) {
    *val = 2 * (*val);
    return 1;
}
```

Our function takes an integer and doubles it. And this is how we call it:

```
int cv = 12;
byrefInt(&cv);
std::cout << cv << "\n";
```

The function has a local variable (*val*) which is a pointer to an integer. This value is initialized with the *address of* the *cv* variable of the calling function. So, when we go `*val = 2 * (*val);` we read and modify the *cv* variable via it pointer *val*.

Efficient passing of large objects

Passing large objects like the array we allocated earlier by creating a copy, or *by value* as is the technical term, is very slow. Instead of copying our data to a new variable it is a lot more efficient to pass a pointer to our data, we actually copy an address and we are done.

We create a function that summarizes the numbers in an array:

```
int sum_data(int* data, int many) {
    int sum = 0;
    for (int i = 0 i < many; ++i) {
        sum += *data;
        ++data;  // point to the next in line
    }
    return sum;
}
```

This function is very flexible. It gets the array and the number of elements we want to include in our calculation. The *many* variable cannot exceed the number of elements we allocated and we have to pass it because the function does not know when to stop.

In the function we modify the pointer to move it to the next item. The pointer belongs to the function, only the data belong to the caller. It is a local function and a *copy* of the caller's pointer.

Here is how we call the function:

```
int s = sum_data(dynPtr, many);
std::cout << "s=" << s << "\n";
```

References

In C++, *references* are essentially aliases for existing variables. They allow you to create another name for a variable, which can be used to access or modify the original variable. *References* were first created to substitute the pointers when used as references to variables as we saw earlier.

References have several advantages over them by design.

Simplified Syntax: References do not require explicit dereferencing, making the code cleaner and easier to read. You can use them just like normal variables.

Memory Efficiency: References do not consume extra memory as they share the same memory address as the variable they refer to. This is unlike pointers, which need additional space for storing the address.

Safety: Since references must be initialized when they are declared, they are less prone to errors like null or wild pointers.

Performance: Passing large objects by reference avoids the overhead of copying the object, which can save both time and memory.

Function Parameters: References allow functions to modify the given parameters directly, which can be useful for functions that need to alter the input variables.

Consistency: Once a reference is initialized to a variable, it cannot be changed to refer to another variable, ensuring consistent behavior.

Creating a Reference

To create a *reference*, we use the *ampersand (&)* symbol:

```
int var;
int& ref = var;
```

The reference *ref* is initialized to the variable *var* right at the declaration. From that point on any changes made to *ref* will be reflected to *var* and vice versa:

```
ref = 14;
std::cout << "var=" << var << "\n";
```

Function parameters

We can use references as function parameters instead of *pointers* when we need to pass a variable by reference or pass large objects like arrays.

```
int byrefVar(int& ref) {
    ref = 24;
    return 1;
}
```

We call the function like this:

```
int var;
byrefVar(var);
std::cout << "var=" << var << "\n";
```

We pass the variable, and the compiler automatically generates the reference the called function expects.

Avoid copying of large objects

We can use references to avoid copying large objects like *std::vectors* of objects. We were introduced to *vectors* earlier. A function that takes vector of integers is like this:

```
void modifyVector(std::vector<int>& v) {
    for (int i = 0; i < v.size(); ++i)
        v[i] += 3;
}
```

The vector is aware of the allocated size, and we no longer need it as argument in the function.

We create and pass to a function like this:

```
std::vector<int> vec{ 1,2,3,4,5,6,7,8,9,10 };
modifyVector(vec);
// view the results
for (int i = 0; i < vec.size(); ++i)
    std::cout << vec[i] << "\n";
```

Key points when using references

Initialization: References must be initialized when declared.

Immutability: Once a reference is bound to a variable, it cannot be changed to refer to another variable.

Computer memory organization

C++ allows us to allocate memory dynamically at runtime. Here we are going to see how C++ organizes computer memory and where everything resides. These are essential for a C++ developer and can make a huge difference in the quality of our programs.

Basic memory organization of a C++ program

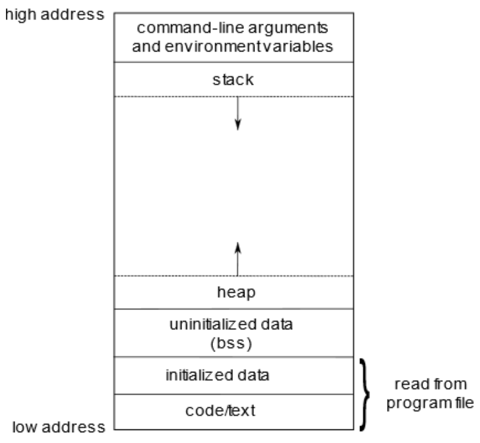
Computer programs are given memory space upon start up. The amount of this memory is really big, theoretically close to the total system memory. This is not physical and reserved for the program though. It is a virtual space managed by the operating system, but our code cannot tell the difference. It communicates with the operating system with compiler vendor supplied libraries and handles all its requirements.

To access this memory we use addresses, aka numbers referring to the location of the data in memory. These addresses start from 0 and go as high as the system supports. Address near the bottom, or 0, is called *low address* and at the top is called *high address*.

In this memory we have everything our program needs.

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.
   Pointers are variables that live on the *stack*, but the memory they point to lives in the *heap*.
6. *Command line arguments and environment variables*. This segment is at the top of the program's address space. The operating system stores this information for our program to use if needed.

Here is a diagram of the memory as described here:



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

Part 5: Strings & Arrays

Arrays

In C++, an array is a data structure used to hold multiple items of the same type in a contiguous memory.

One-dimensional arrays

For example, we could use an array to hold the 10 high scores in a game. The declaration of an array is like this:

```
int hiScores[10];
```

This will allocate a 10-integer space on the *stack*. As we have seen this space will be automatically released when the function ends. We do not have to call *new* to allocate it or *delete* to release it. This memory is not dynamically managed.

We access each element in the array using the indexing operation:

```
for (int i = 0; i < 10; i++)
    hiScores[i] = 0;
```

Array indexing starts at *zero (0)* and goes as high as *number of elements-1*.

Here are the key characteristics of arrays.

- An Array is a collection of data of the same data type, stored at a contiguous memory location.
- Indexing of an array starts from 0. It means the first element is stored at the 0th index, the second at 1st, and so on.
- Elements of an array can be accessed using their indices.
- Once an array is declared its size remains constant throughout the program.
- An array can have multiple dimensions.
- The size of the array in bytes can be determined by the *sizeof* operator using which we can also find the number of elements in the array. Dividing the two gives us the number of elements in the array.

Multi-dimensional arrays

Arrays can have more than one dimensions. In the case of two-dimensional arrays, we can visualize them as grouping made up of rows and columns:

```
int md_hiScores[5][10];
```

This is a high-score table for 5 games and 10 players for each game.

	Column 1	Column 2	Column 3	…
Row 1	md_hiScores[0][0]	md_hiScores[0][1]	md_hiScores[0][2]	…
Row 2	md_hiScores[1][0]	md_hiScores[1][1]	md_hiScores[1][2]	…
…	…	…	…	…

Here is how to iterate over an array with two dimensions:

```
int md_hiScores[5][10];
for (int game = 0; game < 5; game++) {
    for (int player = 0; player < 10; player++) {
        md_hiScores[game][player] = 0;
    }
}
```

Three dimensions is like having multiple sheets and each of them having a two-dimensional table:

```
int hd_hiScores[3][5][10];
```

So now we have:

Sheet 0	Column 1	Column 2	Column 3	…
Row 1	hd_hiScores[0][0][0]	hd_hiScores[0][0][1]	hd_hiScores[0][0][2]	…
Row 2	hd_hiScores[0][1][0]	hd_hiScores[0][1][1]	hd_hiScores[0][1][2]	…
…	…	…	…	…

Sheet 1	Column 1	Column 2	Column 3	…
Row 1	hd_hiScores[1][0][0]	hd_hiScores[1][0][1]	hd_hiScores[1][0][2]	…
Row 2	hd_hiScores[1][1][0]	hd_hiScores[1][1][1]	hd_hiScores[1][1][2]	…
…	…	…	…	…

Sheet 2	Column 1	Column 2	Column 3	…
Row 1	hd_hiScores[2][0][0]	hd_hiScores[2][0][1]	hd_hiScores[2][0][2]	…
Row 2	hd_hiScores[2][1][0]	hd_hiScores[2][1][1]	hd_hiScores[2][1][2]	…
…	…	…	…	…

Here is how we can use a three-dimensional array:

```
int hd_hiScores[3][5][10];
for (int system = 0; system < 3; system++) {
    for (int game = 0; game < 5; game++) {
        for (int player = 0; player < 10; player++) {
            hd_hiScores[system][game][player] = 0;
        }
    }
}
```

Declaration and initialization of arrays

Here is how we declare and initialize arrays.

For one-dimensional arrays we can use these syntaxes:

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

Here is an example:

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

For two-dimensional arrays we have:

```
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} … }
```

And in actual C++ code:

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

Finally for three-dimensional arrays we have:

```
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} … }, …}
```

And in code:

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

Arrays as arguments to functions

In C++ we can use arrays as arguments to functions. Here are the main methods to do it:

Passing as sized array

We can pass an array with a specified size. However, we also need to pass the size of the array as a separate parameter because arrays decay to pointers when passed to functions.

```
void displayArray(int arr[5]) {
    for(int i = 0; i < 5; i++)
        std::cout << "v1 "<<arr[i] << "\n";
}

int main() {
    int arr[]{ 1,2,3,4,5 };
    displayArray(arr);
}
```

Passing as unsized array

We can pass an array without specifying its size in the function parameter. The size is passed separately. This method is more flexible than the previous.

```
void displayArray(int arr[], int many) {
    for (int i = 0; i < many; i++)
        std::cout << "v2 " << arr[i] << "\n";
}

int main() {
    int arr[]{ 1,2,3,4,5 };
    displayArray(arr, 5);
}
```

Passing as a pointer

We can pass the array as a pointer, which is essentially what happens when we pass an array to a function.

```
void displayArray_ptr(int* arr, int many) {
    for (int i = 0; i < many; i++)
        std::cout << "v3 "  << arr[i] << "\n";
}

int main() {
    int arr[]{ 1,2,3,4,5 };
    displayArray_ptr(arr, 5);
}
```

Passing multidimensional arrays

We can also pass multidimensional arrays to functions. However, we need to specify the size of all dimensions except the first, which restricts things a little.

```
void displayArray(int arr[][10], int rows) {
    for (int r = 0; r < rows; r++) {
        for (int c = 0; c < 10; c++) {
            std::cout << arr[r][c] << " ";
        }
        std::cout << "\n";
    }
}

int main() {
    int md_hiScores[5][10];
    for (int game = 0; game < 5; game++) {
        for (int player = 0; player < 10; player++) {
            md_hiScores[game][player] = game * 10 + player;
        }
    }
    displayArray(md_hiScores, 5, 10);
}
```

Arrays using the C++ Standard Library

In *Part 4*we were introduced to *std::vector*, the C++ standard library to dynamically handle arrays. Now we are going to see how we can overcome the limitations of arrays using *vectors*.

Declaration of vector

Vectors are *one-dimensional*. But they can contain any kind of entity as elements. This feature enables us to create multidimensional vectors by specifying vectors containing vectors.

Here is the general syntax of declaring a vector:

```
std::vector<element_type> vec;
std::vector<element_type> vec = { initializer_list };
```

We specify we want to declare a *vector* that will contain elements of *element_type*. We will explain the odd syntax later when we talk about *templates*. For now, we just accept as it is.

Here is a *vector* of integers, and how to initialize if we need:

```
std::vector<int> vec;
std::vector<int> vec = { 1,3,5 };
```

Using a vector

Vector elements are accessed like ordinary array elements and are indexed the same way too, starting from 0:

```
std::vector<int> vec = { 1,3,5 };
std::cout << vec[1] << "\n",
```

A vector has built in dynamic memory management and can grow or shrink. We can add elements at the back or insert them anywhere inside. We can remove any element we want too. The only efficient operations though are adding and removing at the end it. The rest require a lot of memory move and copy operations to keep the internal array in a contiguous memory that make them inefficient.

Here is a list of the most common operations we can do on a vector:

reserve	Reserve initial space

push_back	Append element at the end of the vector
pop_back	Erase last element
clear	Clear all contents
size	Returns the number of elements in the vector

Vectors as function arguments

We can pass a vector as an argument to a function. To make our function efficient we make it accept a *reference* to a vector and not a copy of it.

Here is a complete example of a vector:

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

void printVec(std::vector<int>& vec) {
    for (int i = 0; i < vec.size(); ++i)
        std::cout << i << ". " << vec[i] << "\n";
}

int main() {
    std::vector<int> vec = { 1,3,5 };
    std::cout << vec[1] << "\n";
    vec.push_back(7);
    vec.push_back(9);

    printVec(vec);
}
```

There is a lot of weird syntax when using a vector. We will explain it Part 6 where we will talk about structures and classes.

Create multidimensional array using vectors

We can create a multidimensional array if we create a vector that has vectors as elements:

```
void printVec(std::vector<std::vector<int>> v2d) {
    for (int r = 0; r < v2d.size(); r++) { // over rows
        std::vector<int>& t = v2d[r]; // reference to the row, not copy
        for (int c = 0; c < t.size(); c++) { // over columns
            std::cout << t[c] << " ";
        }
        std::cout << "\n";
    }
}

int main() {
    std::vector<std::vector<int>> v2d;
    for (int r = 0; r < 5; r++) { // create the rows...
        std::vector<int> t; // create a new row
        v2d.push_back(t);  // add it to the 'array'
        for (int c = 0; c < 10; c++) {
            v2d[r].push_back(r * 10 + c); // add elements to the row
        }
    }
    printVec(v2d);
}
```

printVec can go through the *rows* and *columns* using the *size*of the vectors. Using vectors, we are not restricted to having the same number of elements in each row.

From vectors to pointers

Vectors keep their data in a contiguous area. This is the same as an array on some memory we allocate with *new*. Its is easy to obtain a pointer to the vector data and interface with old code we might have, or with code written in the C programming language. Many external libraries expect data to be passed with pointers. One library we use in games and computer simulations is OpenGL for 3D graphics.

A pointer to the first element of a vector is the pointer to the vector data:

```
void displayArray_ptr(int* arr, int many) {
    for (int i = 0; i < many; i++)
        std::cout << "v3 " << arr[i] << "\n";
}

int main() {
    std::vector<int> vec = { 1,3,5 };
    std::cout << vec[1] << "\n";
    vec.push_back(7);
    vec.push_back(9);

    displayArray_ptr(&vec[0], vec.size());
}
```

We can use the *displayArray_ptr* with the vector.

Strings

String is an array of characters. C++ has two ways of handling them. The first comes from C and is called C-sttyle string handling. The other way comes from the *standard library* and is the most efficient and safe way to handle them.

C-Strings

This is how strings are handled in C programming. In C strings are arrays of characters. This is supported in C++ as well with the functionality available in C. These arrays are of type *char* and have to be terminated with a null (\0) character. In computer terms this is 0.

Since it is an array it has some space allocated for the characters. So, we need allocate one more character than the text we want to store for the terminating zero. We can allocate more space than we actually need.

Declaring a C-string

Here are some string declarations:

```
char str_1[] = "C++";
char str_2[4] = "C++";
char str_3[] = { 'C','+','+','\0' };
char str_4[4] = { 'C','+','+','\0' };
char str_5[100] = "C++";
const char* str_6 = "C++";
```

The first five declarations allocate array space and copy the constant strings in that space. The first two allocate space for the terminating 0 although we do not see it. The next two clearly point the array nature of the string. In the fifth we allocate more space than we actually need. Finally, we have a pointer to a string. In this we do not create a copy of the string, but because we have a constant string that will be in the *initialized data segment* we saw in Part 4, it is declared as *const*. Any variable declared as const maintains its contents until the program terminates.

Using a C-string

We can use a C-string to read/manipulate and print text in our programs. In the example we will create a function that counts the digits we have typed when prompted.

```
int count_numerals(char* str) {
    int count = 0;
    while (*str != '\0') {
        if (*str >= '0' && *str <= '9')
            count++;
        str++;
    }
    return count;
}

int main() {
    char txt[100];
    std::cout << "type some text:";
    std::cin >> txt;
    int n = count_numerals(txt);
    std::cout << txt << "," << n << "\n";
}
```

Using C-strings is quite dangerous. They are pointers to the memory and we can allocate/deallocate storage space with *new* and *delete* with all the risks it may have. We must be very carefull when copying one string to another and make sure there is always enough space for the job.

C++ strings

The internal representation of the string may be the same as in C, but the tools for handling them are very safe, robust and easy to use. They were designed to overcome the difficulties generated by the C-style strings.

Like the *vectors* we saw earlier they are handled by the C++ *std::string* class. In Part 6 we will cover *classes* and *object-oriented programming*, but now we are going to see the basics of strings and appreciate the advantages C++ gives us.

Declaring strings

To use the *std::string* class we need to include the <*string*> header. It all starts with variable declaration:

```
std::string str = "C++ is easy";
std::string another_str("C++ is easy");
```

These declare and initialize strings.

Using std::strings

It is easy to assign a new string to a C++ string, add two strings together, check if two strings are equal, and so many more. The string manipulation library is very rich and most of all intuitive to use.

```
std::string str = "C++ is easy";
std::string another_str("C++ is easy");

std::cout << str << "\n";
another_str = "C++ can handle strings";
str = str + " and " + another_str;
std::cout << str << "\n";
if (str == another_str)
    std::cout << "strings are equal\n";
else
    std::cout << "strings are not equal\n";
```

And all this functionality comes with the most efficient memory management.

From std::string to char for C compatibility*

The two key features that set C++ strings apart from C-strings are *Memory Management* and *Functionality*. Like *vectors* we face situations where we need to pass an old-fashioned *char** to library functions. The standard library offers this as well. It provides a function to access the underlying pointer:

```
const char* chr_ptr = str.c_str();
```

We should note here that *c_str* returns a *constant* pointer. This gives us *read-only* access to the characters in the string, protecting us from modifying it to avoid any possible errors.

Summary

In this part we were introduced to some very useful concepts we will need to start organizing data in our programs:

- Arrays in one or more dimensions
- Vectors to replace arrays with the added versatility
- Character strings in C-style
- Strings in the C++ style

Part 6: Enums, Structs & Classes, user-defined types

Enums

Enum (enumeration) is a user-defined type that accepts a limited number of values. They are mainly used to assign names to integral values.

Declaration

Declaration starts with the *enum* keyword followed by the name we want for this type and finally with the list of names for the values enclosed in curly brackets:

```
enum months { jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };
```

The compiler assigns o to the first name in the enum and increases by one until the end of the list. We can assign our own values to the names if we want:

```
enum days { mon = 1, tue = 2, wed = 3, thu = 4, fri = 5, sat = 6, sun = 7 };
```

Usage

After their declaration, enums are valid types and we use them to define variables:

```
months month = sep;
month = jan;
```

Example

Here is a code example for enums:

```
#include <iostream>

enum months { jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };
enum days { mon = 1, tue = 2, wed = 3, thu = 4, fri = 5, sat = 6, sun = 7 };

int month_days(months mon) {
    int days[] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
    return days[mon];
}

const char* month_name(months mon) {
    const char* names[] = { "jan", "feb", "mar", "apr", "may", "jun", "jul", "aug", "sep",
"oct", "nov", "dec"};
    return names[mon];
}

int main() {
    months month = sep;
    month = jan;
    std::cout << month_name(month) << " has " << month_days(month) << " days\n";
}
```

Structures

The C programming language introduced the concept of *structures*. With them we can pack together a group of variables and use them as single entity under one name. it allows us to create complex data types that are more manageable and logically organized. For all the reasons that led to the creation of C++, it has inherited this feature too.

Defining structures in C++

To define a structure, we use the *struct* keyword follower by the name we want to give to this new type and a block of member definitions enclosed in curly brackets:

```
struct structName {
    dataType var1;
    dataType var2;
    ...
};
```

Here is the definition of a *point* in 3D:

```
struct vec3d {
    float x;
    float y;
    float z;
};
```

After a structure is defined, we can use its name as a new type to declare variables.

```
vec3d p1;
vec3d p2 = { 2,3,5 }; // initialize the point
```

Member variables of a structure can also be structures allowing us to represent even more complex entities:

```
struct triangle {
    vec3d p1;
    vec3d p2;
    vec3d p3;
};
```

Using structures

We can access members of structures individually either to read or to modify their value. To access the members of a structure, we use the *dot operator (.)*. If we have a pointer to a structure, we use the *arrow operator (->)* instead. Here's an example of initializing and accessing structure members:

```
// calculate Euclidean distance
double distance(vec3d& p1, vec3d& p2) {
    double dx = p2.x - p1.x;
    double dy = p2.y - p1.y;
    double dz = p2.z - p1.z;
    double l = sqrt(dx * dx + dy * dy + dz * dz);
    return l;
}

int main() {
    triangle t = { {1,1,1},{2,2,2},{3,3,1} };
    double side_length  = distance(t.p1, t.p2);
    std::cout << side_length << "\n";
}
```

When passing structs as arguments we prefer the *by-reference* call since structs are big objects and this way we save both space and time in the call.

Practical Uses of Structures

Structures are particularly useful in C and C++ for several reasons:

- They allow grouping related data together, making the code more organized and readable.

- They are essential for creating more complex data types that aren't natively supported by C++, such as date, time, or custom records.
- Structures are the foundation for building data structures like linked lists, trees, and more.

Limitations of Structures

Despite their obvious usefulness, structures in C have some limitations:

Memory Consumption: Due to member alignment for speed efficiency, structures may consume more memory than necessary.

No Data Hiding: Structure members are accessible from anywhere within the scope, which does not support encapsulation.

In summary, structures provide a way to handle complex data by grouping different types of variables together. They are versatile and form the basis for many advanced data handling techniques in programming. However, they also come with limitations that we need to be aware of when designing our applications.

Classes

Classes expand the concept of *structures* to allow for the creation of *objects*. They enabled encapsulation of data and functions to operate on that data. They have *data hiding* to keep the implementation secure and robust.

Classes are the blueprints of *objects*. On the other hand, *object* is an instantiation of a class. Speaking in terms of programming, an *object* is a variable and *class* is the definition of the form and behavior of that object.

A *class* is a user-defined data type which holds its own data members and member functions that operate on the data. Data members and member functions define the properties and behavior of the objects of a *class*.

Definition of a class

To define a class, we use the *class* keyword follower by the name we want to give to this new type and a block of member definitions enclosed in curly brackets. Classes introduce the concept of *access specifier*. The *access specifier* determines who has access to the members, data and functions that define the objects of the class.

```
class className {
access_specifier1:
    // variables
    // functions

access_specifier2:
    // variables
    // functions
    ...
};
```

The *access specifier* can be one of the keywords: *public*, *protected* and *private*. Here is our point example as a class:

```
class vec3d {
public:
    double x;
    double y;
    double z;

    void move_by(double dx, double dy, double dz) {
        x += dx;
        y += dy;
        z += dz;
    }
};
```

We have added a member function which moves (translates) the point by a specified amount in each axis. So, we do not need to care about the internal representation of a *point* anymore.

The *access specifier* we used is *public*. This makes all the members visible and accessible to everyone. This attribute remains until we change it with another specifier. By default, everything in a class is considered *private*. *Private* members are only accessible to member functions.

From a class to an object

As we said a class is just a description, a variable of its type is the actual object:

```
int main() {
    // declare and initialize a point
    vec3d pt = { 1,1,1 };
    // move it
    pt.move_by(2, 3, 4);
    // directly set a coordinate
    pt.x = 23;
}
```

We can access the members of the class using the *dot operator (.)* when we have a class or a reference to it, and the *arrow operator (->)* when we have a pointer to the class.

In this example we created a vec3d *object*, and then we called a *member function* to modify it, and we also modified a *member variable, or attribute*, directly.

Class Constructors

The *constructor* is a special member function of the class. As the name implies, it *constructs* or *initializes* an object of that class when it is instantiated. The constructor is automatically called by the compiler. The constructor has the same name as the class, has no return type and may have arguments.

```
class ClassName {
access_specifier:
    // constructor
    ClassName() {
        // some code
    }
    // more declarations
};
```

Revisiting our *vec3d* class, we can modify it like this:

```
class vec3d {
public:
    double x;
    double y;
    double z;

public:
    vec3d() {
        x = y = z = 0;
    }
    …
};
```

Whenever we create a vec3d object it is initialized at (0,0,0). The constructor that has no arguments is also called *default constructor* and can be automatically generated by the compiler if we do not write one. The default constructor has no code and performs no initializations to member variables as we did in the example. The above constructor can also be written like this:

```
vec3d() :x(0), y(0), z(0) { }
```

This also initializes the variables to 0. Constructors can be overloaded like any function, and they can also have default arguments, and these arguments can be used to initialize member variables:

```
vec3d(double _x=0, double _y = 0, double _z = 0) :x(_x), y(_y), z(_z) { }
```

This constructor that can be invoked without arguments, because it has default values for all of them, replaces the default constructor.

Usually, constructors are declared as *public*. There are some cases where we declare them *protected* or *private*. That is done when we want to restrict the instantiation of a class and force the use of special mechanisms to do so. We will discuss some application design methods in the last part of this tutorial where this concept will become clear.

The copy constructor

There is special constructor that takes as argument a reference to an object of the same class. The purpose of this compiler is to make the new object a copy of the object we pass:

```
vec3d(const vec3d & org) : x(org.x), y(org.y), z(org.z) { }
```

The keyword *const* in the declaration means that we will not modify the object passed, and if we try to do so the compiler will raise an error.

How to invoke a constructor

Here is how we can instantiate point objects and control the constructor the compiler will call.

```
// default constructor
vec3d pt1;
// initialization constructor
vec3d pt3(1, 2, 3);
// copy constructor
vec3d pt4(pt3);
```

The constructor called is actually determined by the same rules as any overloaded function.

Data hiding due to constructors

Having a constructor allows us to move the *attributes* of the class under the *private* modifier. This will make them inaccessible to code outside the class giving us more freedom to make any changes we like.

Here is how our *vec3d* class could become:

```
class vec3d {
private:
    double x;
    double y;
    double z;

public:
    vec3d(double _x=0, double _y = 0, double _z = 0) :x(_x), y(_y), z(_z) {
        std::cout << "default/initializing constructor\n";
    }
    vec3d(const vec3d & org) : x(org.x), y(org.y), z(org.z) {
        std::cout << "copy constructor\n";
    }
    void move_to(double _x, double _y, double _z) {
        x = _x;
        y = _y;
        z = _z;
    }

    void move_by(double dx, double dy, double dz) {
        x += dx;
        y += dy;
        z += dz;
    }
    double X() {
        return x;
    }
    double Y() {
        return y;
    }
    double Z() {
        return z;
    }
};
```

We can set initial coordinates for our points when we instantiate them, move them around with the *move_to* and *move_by* functions, and read their positions with the *X*, *Y* and *Z* functions.

Any direct modification to the x,y and z values are now prohibited and only available through functions. This ensures that the values are always valid and within the limits we specify when designing our application.

Class Destructors

The *destructor* is another special member function of a class. It is called automatically when an object goes out of scope or deleted with the *delete operator*. Their primary role is to enable us perform cleanup when an object goes out of existence.

This is a good time to recall the *std::vector* class. Any vector object can grow the memory it occupies to store the data we want dynamically. We do not need to clean that memory ourselves because it is done when the object terminates its lifecycle and its destructor is called.

These are the key points of the *destructor*:

- They have the same name as the class but are prefixed with a *tilde (~)*.
- They cannot be overloaded, meaning a class can only have one destructor.
- They do not accept any arguments and do not return any value, not even *void*.
- They are called automatically when an object goes out of scope, the program ends, a block containing local variables ends, or when a delete operator is called.
- They are our last chance for releasing the memory space occupied by the objects.

Here is the destructor for our point class. Here we just display the message that the object goes out of scope:

```
~vec3d() {
    std::cout << "vec3d goes out of scope\n";
}
```

Constructors and Destructors are essential in C++ because they allow us to effectively manage computer resources such as memory we consume during our program execution. This simplifies the design of our software and minimizes the number of errors we may make when programming.

The *this* pointer

Every object is located somewhere in the computer memory and there are the data members. Its member functions though are in another place and their code is shared among all objects. The *this* pointer is a special pointer that allows the functions to operate on the correct data every time. This pointer is implicitly passed to the function by the compiler and each member variable is actually accessed like *this->variableName*, although we write *variableName*.

These are some key uses of the *this* pointer:

Accessing members: although it is used implicitly by the compiler we can also use it explicitly *this->variableName*.

Resolving Name Conflicts: when a function argument has the same name as a member function we can use the *this* pointer to resolve the conflict:

```
void move_to(double x, double y, double z) {
    this->x = x;
    this->y = y;
    this->z = z;
}
```

Returning the object itself: The *this* pointer can be used to return the object itself from a member function, which is useful for chaining function calls:

```
// a function returning a reference to the object
vec3d& move_by(double dx, double dy, double dz) {
    x += dx;
    y += dy;
    z += dz;
    return *this;
}

// moving the point and then getting the coordinate
std::cout << "pt4.x=" << pt4.move_by(5,6,7).X() << "\n";
```

Member functions

C++ introduced the concept of *member functions*. These functions are declared as members of the class like the member variables. They can access the and modify the member variables and define the behavior of the objects.

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

Declaration and Definition of member functions

Member functions can be declared inside the class definition. We can also define member functions outside the class using the *scope resolution operator (::)*. Here is the definition of the *vec3d* class in the header:

```
class vec3d {
private:
    double x;
    double y;
    double z;

public:
    vec3d(double _x = 0, double _y = 0, double _z = 0);
    vec3d(const point& org);
    ~vec3d();
    point& move_to(double x, double y, double z);
    point& move_by(double dx, double dy, double dz);
    double X();
    double Y();
    double Z();
};
```

In the header we only declare the member functions. Their implementation is in the respective source file:

```
vec3d::point(double _x /*= 0*/, double _y /*= 0*/, double _z /*= 0*/) :x(_x), y(_y), z(_z) {
    std::cout << "default/initializing constructor\n";
}
vec3d::point(const vec3d& org) : x(org.x), y(org.y), z(org.z) {
    std::cout << "copy constructor\n";
}
vec3d::~vec3d() {
    std::cout << "point goes out of scope\n";
}
vec3d & vec3d::move_to(double x, double y, double z) {
    this->x = x;
    this->y = y;
    this->z = z;
    return *this;
}
vec3d & vec3d::move_by(double dx, double dy, double dz) {
    x += dx;
    y += dy;
    z += dz;
    return *this;
}
double vec3d::X() {
    return x;
}
double vec3d::Y() {
    return y;
}
double vec3d::Z() {
    return z;
}
```

Using member functions

To use a member function, we create an object and then call the function using the name of the object with the *dot operator (.)*. the member function will be invoked with the right object data.

```
vec3d my_point(2, 3, 1);
my_point.move_to(5, 6, 2);
```

If we have a pointer to the object then we use the *arrow operator (->)*.

```
void someFunction(vec3d * pt) {
    pt->move_by(2, 2, 2);
}
```

Member functions & the const qualifier

A *const* member function is a functions that does not modify the object or call any *non-const* member function. This situation arises when we declare a *const object* variable. In this example:

```
const vec3d c_point(4, 4, 4);
double x = c_point.X();
```

We get a compilation error because we declared *c_point* as *const*, and there is no guarantee that calling *X()* to read the x-coordinate does not modify the internal state of the object. To overcome this, we must make the function *X()* as *const* by appending the *const* qualifier after the argument list both in the declaration and the definition:

```
// the declaration of the function
double X() const;

// and its definition
double vec3d::X() const{
    return x;
}
```

Overloading member functions

Member functions can be overloaded just like any other function in C++. Apart from the argument list, member functions can also differ in the *const* qualifier:

```
// the declarations
double X() const;
double X();

// and the definitions
double vec3d::X() const{
    std::cout << "const X()\n";
    return x;
}
double vec3d::X() {
    std::cout << "non const X()\n";
    return x;
}
```

These are validly overloaded functions although they have the same argument list. The first is called when we have a *const* object and the second is called when we have an ordinary object.

Member variables

Member variables are variables declared inside a class. They are the attributes of the objects created from the class. Member variables can be under any access specifier.

Provided that they are *public*, we can access them using the *dot operator (.)*, or in case we have a pointer using the *arrow operator(->)*. If they are not *public* and we try to access them we will get a compilation error. Using the *vec3d* class, we created before we could make the following mistake:

```
void someFunction(vec3d * pt) {
    pt->x = 10;  // generates compilation error
}
```

Static members

We can declare class members as *static*. This applies both to variables and functions.

Static member variables

Static member variables belong to the class itself and not to the objects. This means they are no copied for every object but are shared among all objects of the class.

Static member variables are declared in the class definition. They are assigned storage space in program memory when they are defined in a source file, usually the file we define the code of the class. Here is the definition of our *vec3d* class with a new *static* variable.

```
class vec3d {
private:
    // other declarations
    static int counter;
};
// and in the source file we add this declaration
int vec3d::counter = 0;
```

It is a good practice to initialize our variables when we declare them.

Our member functions have access to this variable:

```
vec3d::vec3d(double _x /*= 0*/, double _y /*= 0*/, double _z /*= 0*/) :x(_x), y(_y), z(_z) {
    std::cout << "default/initializing constructor\n";
    ++counter;
}
vec3d::vec3d(const vec3d & org) : x(org.x), y(org.y), z(org.z) {
    std::cout << "copy constructor\n";
    ++counter;
}
vec3d::~vec3d() {
    std::cout << "vec3d goes out of scope\n";
    --counter;
}
```

The constructors increase the value of the variable and the destructors decrease it, thus making it a counter of how many objects of the class exist.

Static member functions

Static member functions also belong to the class itself and not to the objects. They do not have access to *this* pointer and cannot call any ordinary member function or access any data member.

However, they can access the private members of any object of the class we pass them as arguments.

They are declared with the keyword *static* preceding the function declaration. Their definition can be done in the class definition or out of it like ordinary functions.

We can call them using the name of the class followed by the scope operator (::) and the function name. Here is an example:

```
class vec3d {
    // other definitions
public:
    // defining the static function
    static int get_counter() {
        return counter;
    }
};

// and calling it
std::cout << "active objects:" << vec3d::get_counter() << "\n";
```

Operator overloading

Operator overloading allows us to redefine how operators work. They are useful when we want to define operations involving user-defined types like classes.

Assume we have to add two *vec3d* objects. Our only option is to write a function that takes two vec3d objects and return their sum.

```
inline vec3d add_vectors(const vec3d& v1, const vec3d& v2) {
    return vec3d(v1.X() + v2.X(), v1.Y() + v2.Y(), v1.Z() + v2.Z());
}
```

The code could be like:

```
vec3d a1(1, 2, 3);
vec3d a2(4, 5, 6);
vec3d a3 = add_vectors(a1, a2);
```

This solution is acceptable but not very elegant. We can redefine the *addition operator (+)* to call a function we create and implement the vector addition as we know it. So, when the compiler encounters an addition of two vectors it will call that function automatically.

Here is the syntax of *operator overloading*:

```
class ClassName {
public:
    returnType operator operator_symbol (argument_list) {
        // code
        return returnType;
    }
};
```

In our case we added the operator in the *vec3d* class:

```
vec3d operator+(const vec3d& v2) {
    return vec3d(x + v2.x, y + v2.y, z + v2.z);
}
```

And our code will become:

```
vec3d a1(1, 2, 3);
vec3d a2(4, 5, 6);
vec3d a3 = a1 + a2;
```

Which is a lot more intuitive and easier to understand.

Structures revisited

In C++ structures are identical to classes. Whatever we have said about classes is also meant for structures as well. There is one difference only. The default *access level* for structures is *public* unless we modify it, while for classes it is *private*. The main reason is to assist porting of C code to C++ and to assist C developers move to C++.

It is a good practice not to modify *access* for *structs* and use *class* if you want to take advantage of the object-oriented features C++ has. This will make your code easier to read and understand.

Function objects

Function Objects or *functors* are class or structure objects that can be called like a function. They actually extend the *function pointers* we saw in Part3. This is done by overloading the *function call operator ()*. Being classes or structures makes them far more flexible than simple function pointers.

Create a function object

All we have to do is define the *function-call operator*:

```
class is_odd {
public:
    is_odd() {}
    bool operator()(int value) {
        if (value % 2) return true;
        return false;
    }
};
```

Here we create a function object that checks if a number is odd. We can create an instance of this class and use it to evaluate some numbers:

```
int main() {
    is_odd iso;

    for (int i = 0; i < 10; i++) {
        if (iso(i))
            std::cout << i << " is odd\n";
    }
}
```

This functor can be extended to handle both odds and evens:

```
class odd_or_even {
    int chk;
public:
    odd_or_even(int c) :chk(c % 2) {}
    bool operator()(int value) {
        if (value % 2 == chk) return true;
        return false;
    }
};
```

The behavior depends on the value we pass to the constructor at instantiation. Odd numbers make it identify odds and even numbers the evens:

```
odd_or_even odds(1);  // with odd numbers it checks for odds
odd_or_even evens(0); // with even numbers it checks for evens
```

We can even pass it to another function or class in order to dynamically modify its behavior:

```
// the template enables the function to accept both function pointers and functors
template<typename F>
void check_values(std::vector<int>& vals, std::vector<int>& results, F& check_fun) {
    for (auto i : vals) {
        if (check_fun(i)){
            results.push_back(i);
        }
    }
}
```

The function is defined as template so that it can adapt to whatever argument we use, as long as it implements the *function call operator*. This allows us to use *function pointers* as well:

```
std::vector<int> results;
check_values(values, results, odds);

results.clear();
check_values(values, results, is_odd);
```

Lambdas

Lambda expressions in C++ are a feature introduced in C++11 that allow us to define anonymous function objects directly in our code. They are particularly useful for short snippets of code that you want to pass to algorithms or asynchronous functions.

He syntax of lambda expressions is:

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

- [capture_clause]: is the capture_clause or lambda introducer. It marks the start of the lambda expression. It can be empty.
- (parameters): is the parameters list, the same as in every function.
- return_type: 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.
- { statement(s) }: the code of the function.

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.

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

Here is an example:

```
// 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 lambda_s()
{
    // basic lambda, we create an inline function
    auto greet = []() {std::cout << "lambda sample\n"; };
    greet();

    std::vector<int> v= { 1, 8, 3, 4, 0, 9, 7, 2, 1, 3, 5, 6,
                          3, 4, 7, 2, 1, 8, 5, 6, 3, 9, 7, 2 };
    // 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 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";
}
```

Summary

In this part we covered:

- Structures
- Classes
- Function objects
- Lambda functions

Part 7: Inheritance

Inheritance is the capability of one class to inherit the attributes and behavior of another class. It is one of the most important features of *Object-Oriented Programming*.

The class that acts as base for the inheritance is called *base class* or *parent class*.

The new class is called *derived class* or *child class*.

Defining a derived class

The basic syntax for deriving a class from another class is:

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

The *access_specifier* in combination with the *access_specifiers* used in the declaration of the *base class* determines what the *derived class* can *directly access* from the *base class*. We will cover this later.

Here is an example of class derivation:

```
// base geometry shape
class geom_shape {
};

// a triangle is a shape
class tria : public geom_shape {
};

// a quad is also a shape
class quad : public geom_shape {
};
```

The basic concept of *inheritance* is the *is-a* relationship between *parent-child* classes. The child class is a specialization of the parent class. Although it is the same as the parent it adds some characteristics that make it different from other types. A triangle and a quadrilateral are both geometric shapes and thus share some characteristics, but they also have some distinct attributes that make them different.

In this example it becomes clear how inheritance works:

```cpp
#ifndef __inheritance_h__
#define __inheritance_h__

enum g_type { t_tria, t_quad };

class geom_shape {
    g_type m_type;
public:
    geom_shape(g_type t):m_type(t) {
    }
    g_type type() const {
        return m_type;
    }
    int num_sides() {
        switch (m_type){
        case t_tria:
            return 3;
        case t_quad:
            return 4;
        }
        return 0;
    }
};

class tria : public geom_shape {
public:
    tria() : geom_shape(t_tria) {
    }
};

class quad : public geom_shape {
public :
    quad() : geom_shape(t_quad) {
    }
    int num_diagonals() {
        return 2;
    }
};

#endif // __inheritance_h__
```

Here we have some functionality in the base class and some specialized in the derived.

The constructor of the base class accepts the type of the object to store it in the member variable *m_type*. Then it uses this type to answer the number of sides an object has. The derived classes have no arguments in their constructors, but they call the base class constructor passing their respective types. This is the first of the differentiations we have. Apart from that the *quad* class gives an extra option to query the number of diagonals an object has.

Modifying behavior

The derived class inherits attributes and behavior from parent. We can have a function in the derived class with the same prototype as in the base class. This allows us to change the behavior of the class. In the above example we could rewrite the *num_sides* function in each class and let it return the correct number:

```
class geom_shape {
public:
    int num_sides() {
        // default returns 0
        return 0;
    }
};

class tria : public geom_shape {
public:
    int num_sides() {
        return 3;
    }
};

class quad : public geom_shape {
public :
    int num_sides() {
        return 4;
    }
};
```

Our calling code remains unchanged. Only the class definitions changed. This allows us to define different behavior for every class when we need.

Virtual functions

Using a pointer or a reference to access an object is something we have seen in before. Inheritance is creating another use case requiring a different solution. A derived class IS-A base class too. So, we can access it through a pointer or a reference to the base class. The following is valid code:

```
geom_shape* ptr = new quad;
```

Having a pointer of the base class and assign it a pointer of the derived class. The question is what would happen if we called *num_sides* using this pointer?

```
std::cout << "sides of ptr=" << ptr->num_sides() << "\n";
```

This calls the base class version of the program despite the fact that it is a *quad*.

C++ introduced the concept of *virtual functions*. This allows us to overcome the problem we just encountered and call the correct function regardless of the type specified by the pointer.

Syntax

The syntax of the *virtual function* declaration is very simple. We just have to prepend the keyword *virtual* in the start of the function declaration.

```
virtual returnType function(argumentList);
```

How it works

Every object has a so-called *virtual table*. There the compiler stores the addresses of the virtual functions. So, when it generates the call it first checks the virtual table and if the function is in it the compiler uses it for the call. If the function is not there, the compiler selects the function based on the object type and when we have a base class pointer it calls the base class function.

Example

We have modified the code and now it calls the function we expect it would:

```
class geom_shape {
    virtual int num_sides() {
        // default returns 0
        return 0;
    }
};

class tria : public geom_shape {
    virtual int num_sides() {
        return 3;
    }
};

class quad : public geom_shape {
public :
    }
    virtual int num_sides() {
        return 4;
    }
};
```

Here is the output when running the code:

```
type of q=1      -- quad
sides of q=4     -- direct call
type of t=0      -- tria
sides of t=3     -- direct call
sides of ptr=4   -- calling quad through a pointer to the base class!
```

Virtual destructors

When we use pointers of the base class to store pointers to objects of a derived class it is good to have *virtual destructors*. This way if we need to call *delete* to destroy an object the compiler will generate the correct call.

```
virtual ~geom_shape() {}
```

Calling base class functions

When we need to call a base class function, usually a virtual function, we can do so using the base class name followed by the *scope operator (::)* and the name of the function.

In virtual functions we may need to call the same function from the base class to perform default actions. Then we may proceed with the derived class specialization:

```cpp
class geom_shape {
public:
    virtual void print() {
        std::cout << m_type;
    }
};

class tria : public geom_shape {
public:
    virtual void print() {
        geom_shape::print();
        std::cout << " tria\n";
    }
};

class quad : public geom_shape {
public :
    virtual void print() {
        geom_shape::print();
        std::cout << " quad\n";
    }
};
```

If we call the *print* function on any of the derived classes, it will first call the base class to do the default and then do its own stuff. This is very useful in big and complex classes and allows us to reuse code instead of rewriting it.

Abstract classes

Abstract classes are designed to act as interfaces and cannot be instantiated directly. This means we cannot create any object of this type. They can only be accessed via pointer to a derived class object.

How to create an abstract class

A class the contains at least one *pure virtual* function is an *abstract class*. A *pure virtual* function is a virtual function that has no body but is assigned to zero (0) when declared like this:

```cpp
virtual int num_sides() = 0;
```

This inserts a NULL pointer in the *virtual table*, which is not allowed for class instances, objects, and so we cannot directly create an object of the class.

These are the basic characteristics for *abstract classes*:

No Instances: We cannot instantiate any objects of an abstract class.

Pointers and References: We can have pointers and references to an abstract class.

Derived Classes: A derived class must override the *virtual* function, otherwise it becomes abstract as well

Constructors/Destructors: Abstract classes can have constructors and destructors.

Use of abstract classes

An abstract class is useful when w need to create an abstract generalization to describe a set of objects. In our example we have *trias* and *quads* which have the abstract type *geom_shape*. It is obvious that having an object of the type *geom_shape* should be avoided because it is too general to be useful.

The base class can still be used to hold the common attributes and behavior for the derived classes as well as declare functions that can be shared among derived classes but perform according to the specific object.

Here are the classes with virtual and pure virtual functions:

```
class geom_shape {
    g_type m_type;
public:
    geom_shape(g_type t):m_type(t) {
    }
    g_type type() const {
        return m_type;
    }
    virtual int num_sides() = 0;
};

class tria : public geom_shape {
public:
    tria() : geom_shape(t_tria) {
    }
    virtual int num_sides() {
        return 3;
    }
};

class quad : public geom_shape {
public :
    quad() : geom_shape(t_quad) {
    }
    virtual int num_sides() {
        return 4;
    }
    int num_diagonals() {
        return 2;
    }
};
```

The *geom_shape* cannot be instantiated but it still can act as an interface to any object derived from it:

```
geom_shape* ptr = new quad;
std::cout << "sides of ptr=" << ptr->num_sides() << "\n";
delete ptr;
```

The sample code has remained unchanged. We have a base class pointer but we created a derived class object. Then via this pointer we call the virtual function and the compiler generates the correct call.

From base class to derived class, RTTI

In C++ we can get a derived class pointer when given a base class one. That can be done safely and if the pointer is not of the class we expected it will return us a NULL pointer which we can safely check and avoid a crash.

The mechanism is called *Run-Time Type Information*. This feature allows the type of an object to be determined during the execution of the program.

In C++ we have two mechanisms to retrieve the actual class information of an object. These are *typeid* and *dynamic_cast*.

typeid

The *typeid* operator is used to identify the class of an object at runtime. It returns a reference to a *std::type_info* object, which provides information about the type. Here's an example:

```
geom_shape* ptr = new tria;
std::cout << typeid(*ptr).name() << "\n";
delete ptr;
```

dynamic_cast

The *dynamic_cast* operator is used for down casting pointers or references to more specific types within a class hierarchy. It performs a runtime check to ensure the cast is valid and returns a pointer or reference of the converted type if successful. Here's an example:

```
geom_shape* ptr = new tria;
tria* tptr = dynamic_cast<tria*>(ptr);
if (tptr)
    std::cout << "triangle\n";

quad* qptr = dynamic_cast<quad*>(ptr);
if (qptr)
    std::cout << "quad\n";

std::cout << tptr << ", " << qptr << "\n";
delete ptr;
```

Considerations and Limitations

- *RTTI* is only available for classes that are *polymorphic*, meaning they have *at least one virtual method*.
- The *typeid* operator should not be used on a null pointer, as it will throw a *std::bad_typeid* exception. It is available on any class regardless of virtual functions.
- The *dynamic_cast* operator can only cast to types that are part of the same inheritance hierarchy and will return a null pointer if the cast is not possible.

RTTI provides a powerful tool for managing polymorphic class hierarchies and performing safe runtime type checks and conversions. However, it introduces overhead and can lead to more complex code maintenance.

Access control

Access control is a fundamental concept in *object-oriented* programming. It helps us manage the visibility and accessibility of a class members to its descendants and the rest of the world.

There are three levels of visibility controlled by the *access specifiers*. These are: *public*, *protected* and *private*. They are applied inside a class to limit access to its members and during inheritance to limit how derived class can access the base class.

Access specifiers in a class

Here is how access specifiers modify accessibility inside a class:

1. *Public*: when members are declared as *public* they are accessible from anywhere in the program. Any function or class can access them.
2. *Protected*: protected members can only be accessed by the class members, its descendants and friends. No other class of function can access them.
3. *Private*: private members can be accessed only by member functions and friends. No other class or function, including derived classes can access them.

Access specifiers in inheritance

Inheritance in C++ can be specified as *public*, *protected* and *private*. So far we have seen public inheritance in our examples.

- *public inheritance* maintains access levels between base and derived class.
- *protected inheritance* upgrades access level making the *public* and *protected* members of the base class *protected* in the derived class.
- *private inheritance* makes the *public* and *protected* members of the base class *private* in the derived class.

Here are some examples to clarify things. First we are going to create our base class:

```
class base {
private:
    int priv_member;
protected:
    int prot_member;
public:
    int publ_member;

    base() :priv_member(1), prot_member(2), publ_member(3) {
    }
    void fun_base() {
        std::cout << "base:" << priv_member << ", " << prot_member << ", " << publ_member <<
"\n";
    }
    friend void some_friend(base& b);
    friend class good_friend;
};
```

Public Inheritance.

```
class der_publ :public base {
public:
    der_publ() {}
    void fun_publ() {
        fun_base();
        std::cout << "publ:" << prot_member << ", " << publ_member << "\n";
    }
};
```

The base class members are accessible normally and we can print their values without problem. Now let us see what the rest of the world can access:

```
der_publ pbl;

// all public members of base are accessible
pbl.fun_base();
std::cout << pbl.publ_member << "\n";
```

Members can be accessed as if they were declared in this class.

Protected Inheritance:

```
class der_prot :protected base {
public:
    der_prot() {}
    void fun_prot() {
        fun_base();
        std::cout << "prot:" << prot_member << ", " << publ_member << "\n";
    }
};
```

The derived class continues to have access to members based on their declaration and not the inheritance type. Things change for the rest of the world though:

```
der_prot prt;

// no base functionality is available directly
prt.fun_prot()
```

Now only the derived class functions are visible to the rest of the world.

Private Inheritance:

```
class der_priv :private base {
public:
    der_priv() {}
    void fun_priv() {
        fun_base();
        std::cout << "priv:" << prot_member << ", " << publ_member << "\n";
    }
};
```

Accessibility from the derived class is unchanged.

```
der_priv prv;

// no base functionality is available directly as well
prv.fun_priv();
```

Not much of difference for the rest of the world. *Private* is stronger than *protected* but the result is the same.

Friends: We have declared a *friend function* and a *friend class* in the base class. These have full access to all the members of the class regardless of *access specifiers*:

```
void some_friend(base& b) {
    // we have access in the friend
    std::cout << b.priv_member << "," << b.prot_member << "," << b.publ_member << "\n";
}
class good_friend {
public:
    good_friend(base* b=nullptr){
    }

    void show_all(base* b = nullptr) {
        if (b == nullptr)
            return;
        std::cout << b->priv_member << "," << b->prot_member << "," << b->publ_member <<
"\n";
    }
```

Multilevel Inheritance

Deriving a class from another derived class is called *multilevel inheritance*. The new class inherits attributes and behavior not only by its base class, but also from the class above that. Multilevel inheritance allow us to specialize our classes at each level. Here is a small class hierarchy demonstrating multilevel inheritance:

```
// 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() {
        car::print();
        std::cout << "i am a sports car\n";
    }
};
```

Creating a *sports_car* object will make both *car* and *vehicle* constructors run.

Multiple inheritance

A class can be derived from more than one base class. This is called *multiple inheritance*. It inherits attributes and behavior from all of them. Here is a simple example:

```
// multiple
class base_a{
public:
    base_a() { std::cout << "base_a\n"; }
    virtual void print() {
        std::cout << "base_a\n";
    }
};

class base_b {
public:
    base_b() { std::cout << "base_b\n"; }
    virtual void print() {
        std::cout << "base_b\n";
    }
};

class class_c :public base_a, public base_b {
public:
    class_c() {}
    virtual void print() {
        std::cout << "class_c\n";
    }
};
```

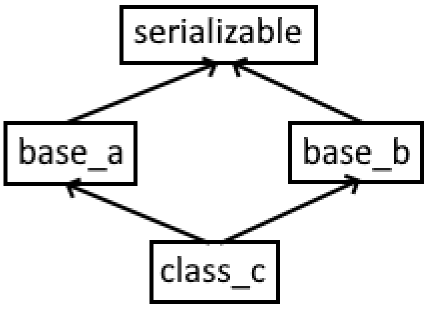
In this example *class_c* is both *base_a* and *base_b*. When we create an object of *class_c* both base classes constructors run.

The diamond problem

This problem occurs when the two base classes have a common ancestor. Here is an example:

```
class serializable {
public:
    serializable() { std::cout << "serializable\n"; }
    void save_to_disk() {}
};
class base_a : public serializable ...
class base_b : public serializable ...
class class_c :public base_a, public base_b ...
```

The next diagram shows the diamond construct.



This is a very common issue with multiple inheritance and can lead to having two instances of the *serializable* class members when we instantiate a *class_c* object. It is easy to see that the constructor of the *serializable* class runs twice, once for *base_a* and one for *base_b*. so we end up with twice as much data which may be crucial in program with big amounts of data.

Virtual Inheritance

The solution to the problem is *virtual inheritance*. This is achieved using the *virtual* keyword. Here is how:

```
class serializable ...
class base_a : virtual public serializable ...
class base_b : virtual public serializable ...
```

This allows the creation of only one copy of the *serializable* class resolving all issues.

Summary

In this part we covered *inheritance*. Specifically, we covered these subjects:

- What is inheritance
- How to define derived classes
- How to use inheritance to modify behavior
- Virtual functions
- Abstract classes
- Run-Time Type Information
- Access control
- Multilevel inheritance
- Multiple Inheritance

Part 8: 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

Function templates are used to create functions. Here we create a function returning the absolute value of a number. Typically, we would have to write a function for every numerical type: integer, float, double etc. Using templates though we write only one and the compiler generates functions for any type needed. This means that if we do not need it no code will be generated.

```
// template function
template<typename T>
T absolute_value(T a)
{
    if (a >= T(0))
        return a;
    return -a;
}
```

Using the template function is easy:

```
int main() {
    int i = -5;
    int av = absolute_value(i);
    std::cout << av << "\n";
    float f = -5.45;
    float fav = absolute_value(f);
    std::cout << fav << "\n";
}
```

The compiler sees the call, determines the type of the argument and generates the correct function. Then it generates a call to that function.

The compiler generated two overloaded functions but we have only one piece of code to maintain.

Class Templates

The idea of templates extends to classes as well. They are very useful when they define generic functionality independent of the actual data type. When we were talking about arrays and dynamic memory management we used the *vector* class. That is actually a template and upon function declaration we were creating the type of vector we needed, like *std::vector<int>*.

Here is an example of a class template:

```
template <typename T>
class vec2d {
    T x;
    T y;
public:
    vec2d(T _x=T(0), T _y=T(0)) :x(_x), y(_y) {
    }
};
```

This class template can be used to hold a two-dimensional vector:

```
int main() {
    vec2d<int> some_pixel;
    vec2d<float> some_coordinates;
}
```

Pixel coordinates on the screen are integer values, while coordinates on a two-dimensional diagram are decimals, or in C++ terms *float* or *double*. Yet we only have one class to maintain.

Template specialization

Template specialization allows the customization of the template behavior for specific types or conditions.

The *absolute_value* function we wrote earlier will be our example. First we will simplify by changing the return value to *double*.

```
template<typename T>
double absolute_value(T a)
{
    if (a >= T(0))
        return a;
    return -a;
}
```

This function can handle the built-in numerical types of the language, but what if we have a *complex* number? The absolute value for a complex number is its magnitude. C++ allows us to define a special version of the template to handle this:

```
class complex {
public:
    double a, b;
    complex(double _a, double _b) :a(_a), b(_b) {
    }
};

template<>
double absolute_value<complex>(complex a)
{
    return sqrt(a.a * a.a + a.b * a.b);
}
```

Note that this extends the template definition and does not add any code in our program unless we call it:

```
int main() {
    std::cout << absolute_value(2.0001) << "\n";

    complex c(3, 4.5);
    std::cout << absolute_value(c) << "\n";
}
```

Template specialization can be used in classes as well:

```
template<typename T>
class templated {
public:
    T value;
    templated(T v) : value(v) {
        std::cout << "generic implementation\n";
    }
};
template<>
class templated<int> {
public:
    int value;
    templated(int v) : value(v){
        std::cout << "specialized implementation\n";
    }
};
```

The class can differentiate its behavior in the case we have integers:

```
int main() {
    std::cout << absolute_value(2.0001) << "\n";

    complex c(3, 4.5);
    std::cout << absolute_value(c) << "\n";

    templated d(7.2); // uses the generic
    templated i(7);   // uses the specialized
}
```

Template Metaprogramming

Template Metaprogramming is a technique of performing computations at compile time using the template mechanism of C++. By default, the C++ optimizes code by performing calculation at compile time whenever this is possible. This stores the precalculated data and saves significant execution time when the program is running. Take a look at the code:

```
int main() {
    const int i = 3;
    int k = 2 * i;
    int j = 1 << i;
}
```

All these calculations are based on constant values, so it is safe to perform them at compilation time. So, the compiler does just that, and the code is equivalent to:

```
const int i = 3;
int k = 6;
int j = 8;
```

The original code has a big advantage because we will change everything just by changing the value of *i* and let the compiler do the rest.

Using templates, we can make the compiler perform more complex calculations for our constant values and save us significant execution time.

Here is how we can calculate factorials in compile time. The classic implementation of *factorial* is like this:

```
int classic_factorial(int v) {
    if (v <= 1) return 1;
    return v * classic_factorial(v - 1);
}
```

It is a simple *recursive* function that we learn as we learn the basics of programming. This code though performs a call and calculations every time:

```
int main() {
    std::cout << classic_factorial(4) << "\n";
}
```

It is a constant value though and it would be nice to replace it with a real constant that we could easily update. For this we will create a *template struct* and let the compiler do it:

```
// the basic template struct
template<int v>
struct factorial {
    enum { value = v * factorial<v - 1>::value };
};
// specialization required to terminate
template<>
struct factorial<1> {
    enum { value = 1 };
};
```

So now we can write:

```
std::cout << factorial<5>::value << "\n";
```

No matter how many times we use this in our code, or what values we pass to the template, it will always generate constant values, and we will not have to browse through the code to find all references and fix. That is done automatically.

These are the key characteristics of template metaprogramming:

- Allows us to create fast and secure code. It is a very efficient way to generate and update constant values in our code.
- The template can only be used in constant evaluation and not in runtime calculations where we have to use classic programming.

Summary

In this part we have addressed:

- What templates are
- Function templates
- Class templates
- Template specialization
- Template metaprogramming

Part 9: 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 extremely 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 an excessively big topic that cannot be covered in a few lines. 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 an application.

Introduction and general concepts

By the time if this writing the latest official standard of the language is C++23. The C++26 standard is in-progress. The most supported version by the compilers is C++17. C++20 is supported but not an all platforms. Only GCC and MSVC support all the features of the standard.

Standardization is a 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:

```cpp
#include <iostream>

int main() {
    std::cout << "This is the C++ Standard Library";
}
```

We will see some of the basic and most common features of the Standard Library.

Containers

Containers in C++ are objects used to store collections of other objects. They are implemented as *templates* which give them the ability to adapt and store any data type.

Types of containers

There are several types of *containers*:

- **Sequential Containers**: Used to store ordered collections of objects. These include *std::array*, *std::vector*, *std::deque*, *std::forward_list*, and *std::list*. These containers maintain the order of insertion and can be accessed sequentially.
- **Associative Containers**: Implement sorted data structures that can be quickly searched. These include *std::set*, *std::multiset*, *std::map*, and *std::multimap*. They are optimized for fast search and retrieval and maintain their elements in a sorted order.
- **Unordered Associative Containers**: They were introduced in C++11, these containers store elements using hash tables allowing for fast access to individual elements based on their keys. This group includes *std::unordered_set*, *std::unordered_multiset*, *std::unordered_map*, and *std::unordered_multimap*.
- **Container Adapters**: These provide a different interface for *sequential containers* suited for specific tasks. They include *std::stack*, *std::queue* and *std::priority_queue*.

Sequential containers

Here is a brief description of the sequential containers.

std::array

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

```cpp
#include <array>

void arrays() {
    std::array<int, 3> ar = { 1,2,3 };
    for (int i = 0; i < 3; i++)
        std::cout << ar[i] << "\n";
}
```

std::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. They are optimized to insert and remove objects at the end of their storage. They also support adding and removing at random locations, but they are very inefficient, and they should be avoided.

```cpp
#include <vector>

void vectors() {
    std::vector<int> v = { 1,2,3 };
    size_t s = v.size();
    for (int i = 0; i < s; i++) {
        std::cout << v[i] << "\n";
    }
}
```

std::forward_list

std::forward_list is a container that implements the *singly-linked* list data structure. In this data structure every object 'knows' the one that comes after it. They do not require contiguous memory thus they are very fast and efficient in insert and remove operations anywhere in the list. It can only be searched or *iterated* from start to end, aka forward only, and not backward.

std::forward_list cannot be accessed with index like a vector or an array. So, we introduce a C++ feature that allows us to iterate over the objects within a container. The *for* loop allows us to iterate over a container like this: *for (auto var_name : container) {…}*. Here it is in action:

```cpp
#include <forward_list>
void forw_list() {
    std::forward_list<int> l;
    l.assign({ 1,2,3 });
    // iterate over the contents of the list
    for (int i : l) {
        std::cout << i << "\n";
    }
}
```

std::list

The *list* container implements the *double-linked* list data structure. They do not require contiguous memory as well and are also very efficient in insert and remove operations. They require a little more memory than *std::forward_list* in order to support walking up and down the list.

```
#include <list>
void list_s() {
    std::list<int> l;
    l.assign({ 1,2,3 });
    for (int i : l) {
        std::cout << i << "\n";
    }
}
```

std::dequeue

std::dequeue stands for *double-ended queue*. It is a sequential container specifically designed and optimize for fast insertion and removal at both ends. It is not guaranteed to store the data in contiguous memory so accessing them with index is not allowed.

```
#include <deque>
void dequeue_s() {
    std::deque<int> d;
    d.push_back(1);
    d.push_front(2);
    for (int i : d) {
        std::cout << i << "\n";
    }
}
```

Associative containers

Here is the list of the *associative containers*:

std::set

std::set is a container that stores unique elements in a specific order. The order is determined by the *less operator (<)*. The default operator defined by C++ is used to sort the items unless define a custom operator for our objects.

```
#include <set>

void set_s() {
    std::set<int> s;
    s.insert(1);
    s.insert(1); // will be rejected, duplicate
    s.insert(2);
    for (int i : s) {
        std::cout << i << "\n";
    }
}
```

std::multiset

std::multiset is similar to std::set, only this allows duplicate values.

```
#include <set>

void multiset_s() {
    std::multiset<int> s;
    s.insert(1);
    s.insert(1); // duplicates are accepted
    s.insert(2);
    for (int i : s) {
        std::cout << i << "\n";
    }
}
```

std::map

std::map is a very powerful associative container that stores data in *key-value pairs*. Here are the key characteristics of maps:

1. Sorted: *std::map* maintains its elements sorted based on the keys. The less operator is the default for sorting but we can customize it.
2. Unique keys: duplicate keys are not allowed in maps.

We can access elements in a map using the *bracket operator []*.

```
#include <map>

void map_s() {
    std::map<int, int> m;
    m.insert(std::pair<int, int>(2, 1));
    m.insert(std::pair<int, int>(3, 10));
    m.insert(std::pair<int, int>(1, 11));

    std::cout << m[2] << "\n";
    for (auto e : m) {
        std::cout << e.first << "," << e.second << "\n";
    }
}
```

Here are some common operations on maps.

Insert: *m.insert(std::pair<key_type, value_type>(key, value));* or *m[key] = value;*

Remove: *m.erase(key);*

Lookup: *m.find(key);* or *m.count(key);*

std::multimap

std::multimap is similar to the *std::map*, only this allows for multiple values under the same key. Another difference is the bracket operator is not supported for multimaps.

```
#include <map>

void multimap_s() {
    std::multimap<int, int> m;
    m.insert(std::pair<int, int>(2, 1));
    m.insert(std::pair<int, int>(3, 10));
    m.insert(std::pair<int, int>(1, 11));
    m.insert(std::pair<int, int>(1, 17));

    for (auto e : m) {
        std::cout << e.first << "," << e.second << "\n";
    }
}
```

Unordered associative containers

Keeping the *associative containers* ordered costs in insert and remove efficiency. To compensate for that in cases we do need the elements to be ordered but we have many insert and remove operations we can use the *unordered associative containers*.

std::unordered_set

std::unordered_set is used to store unique elements in no particular order.

```cpp
#include <unordered_set>

void unordered_set_s() {
    std::unordered_set<int> s;
    s.insert(2);
    s.insert(1);
    s.insert(1); // will be rejected, duplicate
    s.insert(6);
    s.insert(4);
    for (int i : s) {
        std::cout << i << "\n";
    }
}
```

std::unordered_multiset

std::unordered_multiset is like *std::unordered_set* only this one allows for duplicate entries.

```cpp
#include <unordered_set>

void unordered_multiset_s() {
    std::unordered_multiset<int> s;
    s.insert(2);
    s.insert(1);
    s.insert(1); // will be rejected, duplicate
    s.insert(6);
    s.insert(4);
    for (int i : s) {
        std::cout << i << "\n";
    }
}
```

std::unordered_map

std::unordered_map is the unordered counterpart of *std::map*. It sacrifices retrieval speed (only by a margin) and order for faster insert and removal. It maintains the unique feature of the keys and ease of use of std::map.

```cpp
#include <unordered_map>

void unordered_map_s() {
    std::unordered_map<int, int> m;
    m.insert(std::pair<int, int>(2, 1));
    m.insert(std::pair<int, int>(3, 10));
    m.insert(std::pair<int, int>(3, 15)); // will fail previous, no duplicates
    m.insert(std::pair<int, int>(1, 11));
    m[2] = 16;    // will replace previous, no duplicates
    for (auto i : m) {
        std::cout << i.first << ", " << i.second << "\n";
    }
}
```

std::unordered_multimap

std::unordered_multimap is similar to *std::unordered_map* with he exception that this allows duplicate entries. The bracket operator is not supported like *std::multimap*.

```
#include <unordered_map>

void unordered_multimap_s() {
    std::unordered_multimap<int, int> m;
    m.insert(std::pair<int, int>(2, 1));
    m.insert(std::pair<int, int>(3, 10));
    m.insert(std::pair<int, int>(3, 15));
    m.insert(std::pair<int, int>(1, 11));
    for (auto i : m) {
        std::cout << i.first << ", " << i.second << "\n";
    }
}
```

Container Adapters

Sequential and *associative* containers have all the functionality we need. *Unordered associative* containers add the efficiency we may need in some cases. There are times we need to have some containers that put some constraints. These are the *container adapters*. They are based on container already defined and they enforce the restrictions we need. These container adapters are *stack*, *queue* and *priority_queue*. They are meant for storage and retrieval only and do not allow iterators or random access.

std::stack

std::stack implements the classic stack data structure. The last item inserted is the first retrieved (Last-In-First-Out or LIFO).

```
#include <stack>

void stack_s() {
    std::stack<int> s;
    s.push(2);
    s.push(3);
    s.push(1);
    while (!s.empty()) {
        // retrieve last element
        std::cout << s.top() << "\n";
        // and remove it
        s.pop();
    }
}
```

std::queue

std::queue implements the queue data structure. In this the first element inserted is the first retrieved (First-In-First-Out or FIFO).

```
#include <queue>

void queue_s() {
    std::queue<int> s;
    s.push(2);
    s.push(3);
    s.push(1);
    while (!s.empty()) {
        // retrieve first element
        std::cout << s.front() << "\n";
        // and remove it
        s.pop();
    }
}
```

std::priority_queue

std::priority_queue is almost the same as *std::queue* only this one keeps elements *ordered*, hence the *priority* in the name.

```
void priority_queue_s() {
    std::priority_queue<int> s;
    s.push(-1);
    s.push(2);
    s.push(3);
    s.push(1);
    s.push(9);
    while (!s.empty()) {
        // retrieve first element
        std::cout << s.top() << "\n";
        // and remove it
        s.pop();
    }
}
```

Iterators

The iterators are objects that help us traverse through the elements of a container and access them.

Types of iterators

Iterators are classified based on their functionality.

Input Iterators: These are used to read elements from a container in a single-pass algorithm.

Output Iterators: These are used to write elements to a container in a single-pass algorithm.

Forward Iterators: These can read and write elements and can move forward through the container.

Bidirectional Iterators: These are like forward iterators, plus they can move backward through the container.

Random-Access Iterators: These are the most powerful iterators, allowing direct access to any element in the container, similar to a pointer.

Basic usage

Iterators are based on the container they access. Here is how we declare an iterator to a vector:

```
std::vector<int> v={ 1,4,5,3,8,7,6 };
std::vector<int>::iterator it;
```

The container provides methods to initialize the iterator, in the case of the *std::vector* we have the *v.begin()*, which returns an iterator to the first element.

The iterator marches using the *increment operator ++*. We can check if we have reached the end of the vector using the *v.end()* member function. This returns one past the last element. Here is how we go through the element of the vector.

```
void iterators_s() {
    std::vector<int> v={ 1,4,5,3,8,7,6 };

    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
        std::cout << *it << "\n";
}
```

Benefits of Iterators

Convenience: Iterators abstract the details of the container, making it easier to write generic algorithms.

Flexibility: They allow algorithms to work with different types of containers without modification.

Safety: Iterators provide bounds checking, reducing the risk of accessing invalid memory.

Algorithms

C++ offers a rich set of algorithms through the Standard Library, which provides functions for sorting, searching, and manipulating data structures like arrays and vectors. 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 algorithms library relies heavily on function objects. That is our chance to inject our logic and modify the behavior of the algorithms.

Here are some commonly used algorithms:

Sorting

This group of algorithms sorts a given range of values, based on the iterators we pass.

std::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 an example:

```
#include <algorithm>
#include <vector>

void sort_s() {
    std::vector v = { 1,4,8,2,-3,7,9,-5,5,20,6 };
    // the lambda function determines ascending or descending order
    std::sort(v.begin(), v.end(), [](int a, int b) {return a < b; });
    for (auto i : v) {
        std::cout << i << "\n";
    }
}
```

std::stable_sort

This is almost the same as *sort*. The only difference is that if two elements are equivalent it does not swap them.

std::partial_sort

This function can sort only a part of the container. Here is the definition of the algorithm:

```
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 in no particular order.

first: the start of the range to affect.

last: the end of the range.

middle: the element within the range *[first, last)* that is used for the upper boundary of the sort.

The algorithm starts sorting in ascending order the elements in the *[first, last)* range until it has sorted the elements up the *middle* iterator.

In the following sample *partial_sort* will sort the elements of the vector until it has reached the third element, the sorted elements will be *[first, middle)*.

```
void partial_sort_s()
{
    std::vector<int> v = { 10, 8, 5, 4, 3, 19, 0, 9, 7 };
    std::partial_sort(v.begin(), v.begin() + 3, v.end());
    for (auto i:v)
        std::cout << i << ", ";
    std::cout << "\n";
}
```

Searching algorithms

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.

std::find

std::find searches a given *range* in a container to find the first occurrence of a *value*. It returns an iterator at its position if found and the iterator marking the end of the search range if not found.

```
void find_s() {
    std::vector<int> v = { 10, 8, 5, 4, 3, 19, 0, 9, 7 };
    auto f = std::find(v.begin(), v.begin()+5, 10); // search within the first five elements
    if (f == v.begin() + 5) // the end of the range reached
        std::cout << "not found\n";
    else
        std::cout << "found:" << *f << " at position:" << f - v.begin() << "\n";
}
```

std::find_if and std::find_if_not

std::find_if and *std::find_if_not* are similar functions only they give us the opportunity to define a condition in the search instead of value.

```
void find_if_s() {
    std::vector<int> v = { 10, 8, 5, 4, 3, 19, 0, 9, 7 };
    // find the first number bigger than 15
    auto f = std::find_if(v.begin(), v.end(), [](int i) { return i >= 15; });
    if (f == v.end()) // the end of the range reached
        std::cout << "not found\n";
    else
        std::cout << "found:" << *f << " at position:" << f - v.begin() << "\n";
}
```

std::find_if_not is the same with inverted logic (NOT). In simple cases like this *lambda* functions are useful.

std::find_first_of

std::find_first_of compares two containers. Its purpose is to find the first occurrence of any element of the second container in the first.

```
void find_first_of_s() {
    std::vector<int> v = { 10, 8, 5, 4, 3, 19, 0, 9, 7 };
    std::vector<int> s = { 1, -2, 9 };
    auto f = std::find_first_of(v.begin(), v.end(), s.begin(), s.end());
    if (f == v.end()) // the end of the range reached
        std::cout << "not found\n";
    else
        std::cout << "found:" << *f << " at position:" << f - v.begin() << "\n";
}
```

std::binary_search

std::binary_search is another search algorithm designed to work on sorted containers. It actually uses the *binary search* algorithm which works on sorted data. The simple *std::find* algorithm searches the data sequentially and it is slow on big datasets, so for *std::map*, *std::set* and other sorted containers we prefer *std::binary_search* which is faster.

```
void binary_search_s() {
    std::set<int> v = { 10, 8, 5, 4, 3, 19, 0, 9, 7 };

    auto f = std::binary_search(v.begin(), v.end(), 19);
    if (!f) // the end of the range reached
        std::cout << "not found\n";
    else
        std::cout << "found\n";
}
```

std::lower_bound and std::upper_bound

std::lower_bound and *std::upper_bound* are used in *sorted* containers that allow for multiple entries of the key like *std::multiset* and *std::multimap* or *sorted* vectors and lists. They return:

std::lower_bound: an iterator an iterator to the first occurrence of the search key.

std::upper_bound: an iterator one position past the last occurrence of the search key.

```
void bounds_s() {
    std::vector<int> v = { 1,2,3,4,4,4,4,5,6,7,8,9,10 };

    // find first occurrence of 4
    auto lb = std::lower_bound(v.begin(), v.end(), 4);
    // find last occurrence of 4
    auto ub = std::upper_bound(v.begin(), v.end(), 4);
    for (auto i=lb; i<ub; ++i)
        std::cout << *i << " ";
    std::cout << "\n";
}
```

Copy, move and remove algorithms

These algorithms provide efficient *copy*, *move* and *remove* operations for elements in containers.

std::copy and std::copy_if

These algorithms copy a range of data to a new location. The std::copy_if allows us to define a condition that will determine if a copy is allowed or not:

```
void copy_s() {
    std::vector<int> v = { 1,2,3,4,5,6,7,8,9,10 };
    std::vector<int> l = { 21,22,23,24 };

    std::copy(l.begin(), l.end(), v.begin() + 2);

    for (auto i : v)
        std::cout << i << " ";
    std::cout << "\n";
}

void copy_if_s() {
    std::vector<int> v = { 1,2,3,4,5,6,7,8,9,10 };
    std::vector<int> l = { 21,22,23,24 };

    std::copy_if(l.begin(), l.end(), v.begin() + 2, [](int i) { return (i%2)==0; });

    for (auto i : v)
        std::cout << i << " ";
    std::cout << "\n";
}
```

std::remove and std::remove_if

std::remove and *std::remove_if* remove elements from a container. *std::remove* removes all occurrences of a certain value within a range. *std::remove_if* removes all elements that fulfill a given condition. The functions cannot modify the length of the container so instead of deleting elements they shift them towards the start and return an iterator to the new end of the container. Here is an example:

```cpp
void remove_s() {
    std::vector<int> v = { 1,2,3,4,5,6,4,7,8,9,10 };
    auto it = std::remove(v.begin(), v.end(), 4);
    std::cout << "last after removal (0 based!):" << it - v.begin() << "\n";

    for (auto i : v)
        std::cout << i << " ";
    std::cout << "\n";
}

void remove_if_s() {
    std::vector<int> v = { 1,22,31,40,5,6,7,8,9,10 };
    auto it = std::remove_if(v.begin(), v.end(), [](int i) { return (i % 2) == 1; });
    std::cout << "last after removal (0 based!):" << it - v.begin() << "\n";

    for (auto i : v)
        std::cout << i << " ";
    std::cout << "\n";
}
```

Compile and run these samples to see the shift in the elements within the vector.

std::merge

std::merge merges two sorted containers in a new sorted container. Destination container must have enough space allocated for the operation.

```cpp
void merge_s() {
    std::vector<int> v1 = { 1,3,5,7,9 };
    std::vector<int> v2 = { 2,4,6,8,10 };
    std::vector<int> v3(v1.size() + v2.size()); // preallocate space!!
    std::merge(v1.begin(), v1.end(), v2.begin(), v2.end(), v3.begin());
    for (auto i : v3)
        std::cout << i << " ";
    std::cout << "\n";
}
```

std::min and std::max

std::min and std::max return the minimum and maximum values in a container or a list of elements. It uses the *less operator (<)* but a custom comparison function can be given to customize the algorithm.

```cpp
void min_max_s() {
    auto mn = std::min({ 1,3,9,4,6,10,5,8,7 });
    std::cout << "min:" << mn << "\n";
    // using a custom comparison we can get the opposite result!
    auto mx = std::min({ 1,3,9,4,6,10,5,8,7 }, [](int i, int j) { return (i > j); });
    std::cout << "max:" << mx << "\n";
    auto mx2 = std::max({ 1,3,9,4,6,10,5,8,7 });
    std::cout << "max:" << mx2 << "\n";
}
```

Summary

In this part we were introduced to the C++ Standard Library. We have seen:

- Containers
- Iterators
- Algorithms

Part 10: Input / Output

The input/output part of the C++ Standard Library is bug and important that it requires a dedicated chapter. The only input and output we have seen so far is *std::cin* and *std::cout* used to read some basic input from the keyboard and writing some simple text to the screen. Now we are going to go through the input/output provided by the C++ Standard Library.

std::cin and std::cout

In C++ we call the I/O mechanism *stream* because of the stream of information between our program and 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* in the library:

```
void basic_io_s()
{
    std::string s;
    std::cout << "please type your name:";
    std::cin >> s;
    std::cout << "Hello " << s << "\n";
}
```

The *stream operator* (<<) lets us concatenate one object after the other.

std::stringstream

std::stringstream class brings together strings and streams. It works both for input and for output. We can output to this stream like we do with *std::cout*, and then get the string representation of our output, or read a string as if we are reading from the *std::cin*.

```
void strinstream_s()
{
    // 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) {
        // and put them in a vector
        v.push_back(w);
    }
    // this will be an 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

File streams are used to perform input and output operations on files. Files are used for persistent storage. Here we are going to see what is in the C++ Standard Library that we can use to read and write to files.

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 like 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.

File types

Files are divided in two main categories. *Text files* and *binary files*.

Text files are used for text storage, or human readable representation of the information. *Binary files* contain raw memory contents only the computer can interpret and use.

This organization of information within a file is what we call *file format*.

File operations

These are some of the operations we usually do with files

- We can create a file
- Write data to the file
- Append data to the file
- Modify/update data in a 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 recommended that we do it ourselves.

```
o.close();
```

Writing text to a file

The fact that we have a stream makes this operation quite 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_s()
{
    std::ofstream o;
    o.open("some file.txt", std::ios::out);

    o << "some text";
    o.close();

    o.open("some file.txt", std::ios::app);
    for (int i=0;i<10;i++)
        o << "some more text\n";
    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_s()
{
    std::ifstream is;
    is.open("some file.txt", std::ios::in);

    int count = 0;
    std::string s;
    // using stream operator reads word by word
    // while (is >> s)
    //  this reads line by line
    while (getline(is, s))
    {
        std::cout << s << "\n";
        // depending on the loop we count lines or words
        ++count;
    }
    is.close();
    std::cout << "count:" << count << "\n";
}
```

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

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_binary_s()
{
    std::ofstream o;
    // open the file for binary output
    o.open("some file.dat", std::ios::out | std::ios::binary);
    // write an integer
    int i = 1;
    o.write((char*)&i, sizeof(int));
    //write a character
    char c = '3';
    o.write((char*)&c, sizeof(char));
    // save a text string
    std::string s("some text");
    // first save its length
    i = (int)s.length();
    o.write((char*)&i, sizeof(int));
    // and then the contents
    o.write((char*)s.c_str(), i);
    o.close();
}
```

Reading binary files

When reading from binary files we have to rely on functions instead of stream operators again. TThe function to use for reading is *read*. Here we read the file we created before:

```
void read_binary_s()
{
    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";
}
```

A closer look at these 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 make our program less prone to errors and easier to maintain and debug.

Summary

In this part we covered the Input / Output system in the C++ Standard Library. We were introduced to:

- Streams
- String streams
- File streams
- File types
- Opening and closing files
- Using text files
- Using binary files

Part 11: 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. Sometimes we are dealing with numbers hat lead to numerical errors and other times the program comes across a file it cannot read. All these situations may lead to a crash. C++ has built-in tools that can help us catch these errors and handle them gracefully.

Writing robust code

The first step to error handling is adopting techniques that make our code robust and leave very few chances to errors. Here are some good programming habits.

Reuse code

Always break your code down to small and simple functions. Test them extensively and reuse them in your projects. They can become the building blocks of a stable and robust program. Here is a simple function that calculates the value of a polynomial:

```cpp
double polynomial(const std::vector<double>& coeffs, double x) {
    size_t p = coeffs.size() - 1;  // the maximum power of x
    double val = 0;
    for (auto it = coeffs.begin(); it != coeffs.end(); ++it) {
        val += (*it) * pow(x, p);
        --p;  // as we go the power of x decreases
    }
    return val;
}
```

This could be a part of a mathematics library in a physics simulation program or a game. Extensive use and testing will guarantee that it is a safe piece of code to use.

Use reliable external libraries

As consequence of the previous is the selection of reliable external libraries. The first library we rely on is the *C++ Standard Library*. For 3D graphics we rely on *OpenGL* and its successor *Vulkan*. The dedicated work behind the creation of these libraries and their use by others like you is guarantee that these libraries have been extensively tested and are error free.

Validate your data

Validate your data as early as possible in your code. Make sue that there is nothing there that could lead to a crash. Validating at each step is not so easy and it leads to very slow code. Making sure that our data are clear of unwanted values helps us write simple and efficient code being sure that no error will occur.

Test your code

Design and develop test mechanisms in your code that will run and test it under the harder conditions automatically and report any errors that might come up. Run these tests systematically, every night if possible, to make sure that your code responds as expected under any conditions.

Exceptions

Exceptions in C++, provide a robust way to handle errors that arise during program execution. When a division by zero occurs or a file fails to open the system raises an *exception*. C++ has a built-in mechanism that can catch exceptions. C++ Standard Library creates a programmer friendly interface to this mechanism.

try / catch

try / catch is the building block of exception handling in C++. The basic syntax is like this:

```
try {
    code that might cause problems
}
catch (...) {  // catch all exceptions
    code to perform clean-up if an error occurs
}
```

We place our code that might run into problems in the *try* block. Should anything go wrong the code automatically resumes in the *catch* block. There we usually put code to perform any cleanup before we return from our function.

Here is a simple example:

```
void exception_s()
{
    int i = -1;
    try {
        char* buf = new char[i]; // bad allocation exception
        std::cout << "this should not be seen\n";
    }
    catch (...) {  // catch all exceptions
        std::cout << "simple_exception catches exception" << "\n";
    }
    std::cout << "exception_s terminates gracefully!\n";
}
```

Going one step further we can get some information about the error that occurred:

```
void exception2_s()
{
    int i = -1;
    try {
        char* buf = new char[i]; // bad allocation exception
        std::cout << "this should not be seen\n";
    }
    catch (const std::exception& e) {
        std::cout << "exception2_s catches exception:" << e.what() << "\n";
    }
    std::cout << "exception2_s terminates gracefully!\n";
}
```

Using the generic exception, we catch everything (remember polymorphism and virtual functions) and we can get specific information about the error.

Catching specific exceptions

We can have multiple *catch* blocks in exception handling, so that we can take different actions depending on the error. In the following example the function generates two different errors and uses different *catch* blocks to handle them individually:

```
void throw_some_exception() {
    throw std::bad_array_new_length();  // raise a random exception
}

void exception3_s(int ec) {
    int i = -1;
    try {
        if (ec == 1)
            throw_some_exception();
        else
            char* buf = new char[i]; // bad allocation exception
        std::cout << "this should not be seen\n";
    }
    catch (const std::bad_array_new_length& e) {
        // the code resumes here after the bad_array_new_length exception
        std::cout << "exception3_s specific exception:" << e.what() << "\n";
    }
    catch (const std::exception& e) {
        // the code resumes here after any other exception
        std::cout << "exception3_s general exception:" << e.what() << "\n";
    }
    std::cout << "exception3_s terminates gracefully!\n";
}
```

Throwing custom exceptions

We can create and throw custom exceptions in programs too. Using exceptions to break from situations that may lead to crashes is a good but very difficult technique. It requires very good knowledge of the overall architecture of the application, careful planning and above all sticking to the plan or things will get out of hand very soon with unpredictable results.

```
class custom_exception : public std::exception {
    std::string what_s;
public:
    custom_exception(const char* msg = "this is a custom exception") :std::exception(), 
what_s(msg){

    }
    virtual const char* what() const throw() {
        return what_s.c_str();
    }
};

void custom_s(int ex) {
    try {
        if (ex == 1)
            throw custom_exception();
        else
            throw std::bad_array_new_length(); // raise a random exception
    }
    catch (const custom_exception& e) {
        std::cout << "custom_s specific exception:" << e.what() << "\n";
    }
    catch (const std::exception& e) {
        // the code resumes here after any other exception
        std::cout << "custom_s general exception:" << e.what() << "\n";
    }
    std::cout << "custom_s terminates gracefully!\n";
}
```

Summary

Here we tried to approach the topic of writing stable and robust applications. The main topics were:

- Habits that will lead to better code
- Using exceptions to safeguard our code
- Developing and throwing our custom exceptions

Part 12: The Preprocessor

The Preprocessor was inherited from the C programming language. It parses our code before the compiler and generates the final stream that will be compiled. Using its features we can perform some modifications to our code right before compilation.

Preprocessor directives

Preprocessor directives begin with *hash symbol (#)*. This character must be the first of the line, but not necessarily on the first column. Then follows the directive keyword. Blanc space between the hash symbol and the keyword is allowed. Here is a list of the preprocessor directives:

#include

This is the first preprocessor directive we learn as C/C++ developers. This instructs the preprocessor to *include* or *inject* the contents of another file at the current location. We mainly use it to *include* a *header* file with declarations.

#define

This directive is used to *define* a new name or *macro*. This macro can be used to modify the code the preprocessor outputs. Whenever the preprocessor encounters the macro, replaces it with its value:

```
#define PI 3.1415926

int main() {
    double radius = 1.234;
    double circumference = 2 * PI * radius;
}
```

It is a good way to replace difficult to remember values with easy to read and remember names. If we ever decide to increase the accuracy by adding digits to PI we change the definitions and recompile.

It can be used just to define a macro without giving it a value

```
#define __DEBUG__
```

#ifdef / #ifndef / #else / #endif

Check if a compiler directive has been defined and take appropriate action.

```
#include <iostream>

#define __DEBUG__

int main() {
#ifdef __DEBUG__
    std::cout << "debug enabled\n";
#else
    std::cout << "release enabled\n";
#endif
}
```

We can use this to safeguard our headers so they do not load twice:

```
#ifndef __PREPROC_H__
#define __PREPROC_H__

// this will not lead to recursion!!
// #ifndef in the first line prevents it
#include "preproc.h"

#endif // __PREPROC_H__
```

This is a very common situation in large projects where one header loads another and so on until they make a circle and compilation breaks because it sees the same definitions again.

#error

This directive raises a compilation error and stops the compilation process:

```
#ifndef __PREPROC_H__
#error include preproc.h to proceed
#endif
```

#pagma

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" )
```

Predefined preprocessor macros

The preprocessor has some built-in macros we can access in our code. They can be particularly 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 are:

• std:c++14	201402L
• std:c++17	201703L

This macro is extremely useful when we want to interface C++ with C code:

```
#ifdef __cplusplus
// the following definitions follow the C standard
// note the opening bracket, look for the closing bracket
extern "C" {
#endif
    // function that can be called both from C and C++ code
    void some_function();
#ifdef __cplusplus
    // end of C type definitions
    // after the bracket we have C++ style definitions
}
// C++ definitions can follow (always inside #ifdef/#endif block!)
class some_class {
public:
    some_class() {}
};
#endif
```

Summary

In this part we tried to give a brief description of the preprocessor. A valuable tool that if used correctly can help us solve many problems.