C++ course notes

Mohammad El-Moussawi

2022-2023

Hosted At tenji.org

e-mail : sawe@lesawe.com

Contents

Chapter 1

Basics

1.1 Fundamental data types

Name	Description	Size in bytes	Range
char	Charcter or small integer	1	signed: -128 to 127 unsigned: 0 to 255
short int	Short Integer.	2	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int	Long integer.	4	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value.	1	true or false
float	Floating point number.	4	+/- 3.4e +/- 38 (7 digits)
double	Double precision floating point number.	8	+/- 1.7e +/- 308 (15 digits)
long double	Long double precision floating point number.	8	+/- 1.7e +/- 308 (15 digits)
wchar_t	Wide character.	2 or 4	1 wide character

Wide characters are used mainly to represent non-English or exotic character sets.

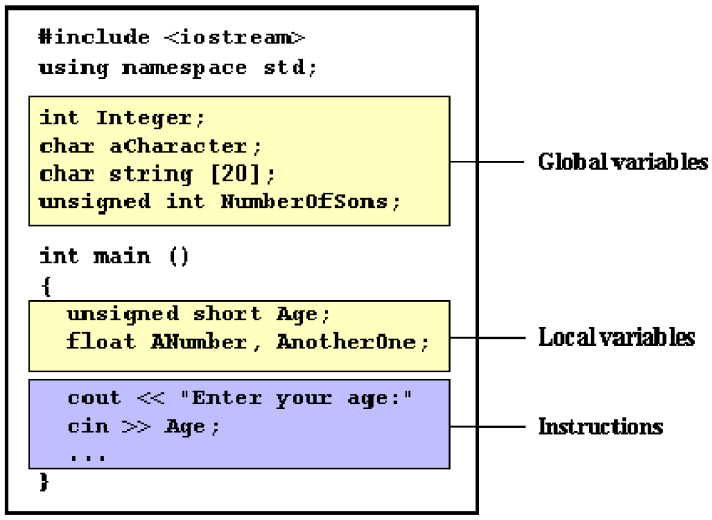
1.2 Declaration and initialization of varaibles

```
int a ;
float mynumber;
int a, b, c; // to declare more than one variable of the same type
unsigned short int Number; // to declare variable with type

int a=5; // initial value = 5
int b(2); // initial value = 2
```

1.3 Scope of variables

A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.



1.4 Intro to strings

We need to include an additional header file in our source code: ¡string¿ and have access to the std namespace .
Declaration and initialization :

```
string mystring = "This is a string";
string mystring ("This is a string");
```

1.5 Constants

Constants are expressions with a fixed value.

1.5.1 Literals

Literals are used to express particular values within the source code of a program.

Integer Numerals

They are numerical constants that identify integer decimal values.
In addition to decimal numbers (base 10) C++ allows the use as literal constants of octal numbers (base 8) and hexadecimal numbers (base 16).

```
75 // decimal
0113 // octal start with 0
0x4b // hexadecimal start with 0x
```

Literal constants, like variables, are considered to have a specific data type.

```
75 // int
75u // unsigned int
75l // long
75ul // unsigned long
```

Floating Point Numbers

They express numbers with decimals and/or exponents.

```
3.14159 // 3.14159
6.02e23 // 6.02 x 10^23
1.6e-19 // 1.6 x 10^-19
3.0 // 3.0

3.14159L // long double
6.02e23f // float
```

Character and string literals

There also exist non-numerical constants, like:

```
'z'
'p'
"Hello world"
"How do you do?"
```

Character and string literals have certain peculiarities, like the escape codes.
These are special characters that are difficult or impossible to express otherwise in the source code of a program:

\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\b	backspace
\f	form feed (page feed)
\a	alert (beep)
\'	single quote(')
\"	double quote(")
\?	question mark (?)
\\	backslash(\)

- String literals can extend to more than a single line of code by putting a backslash sign (\) at the end of each unfinished line.

```
    "string expressed in \
two lines"
```

- You can also concatenate several string constants separating them by one or several blank spaces, tabulators, newline or any other valid blank character:

```
    You can also concatenate several string constants separating them by one or several blan
newline or any other valid blank character:
```

- Finally, if we want the string literal to be explicitly made of wide characters (wchar_t), instead of narrow characters (char), we can precede the constant with the L prefix:

```
    L"This is a wide character string"
```

Boolean literals

There are only two valid Boolean values: true and false. These can be expressed in C++ as values of type bool by using the Boolean literals true and false.

1.5.2 Defined constants

ou can define your own names for constants that you use very often without having to resort to memory-consuming variables, simply by using the #define preprocessor directive. Its format is:

```
#define PI 3.14159
#define NEWLINE '\n'
```

1.5.3 Declared constants

With the const prefix you can declare constants with a specific type in the same way as you would do with a variable:

```
const int pathwidth = 100;
const char tabulator = '\t';
```

1.6 Operators

1.6.1 Assignemnt (=)

The assignment operator assigns a value to a variable.
The assignment operation always takes place from right to left, and never the other way.

```
a=5;
a=b;
```

the assignment operation can be used as the rvalue (or part of an rvalue) for another assignment operation.

```
a = 2+(b=5);
```

is equivalent to :

```
b =5 ;
a = 2 + b;
```

you cans assign a value to multiple variables :

```
a = b = c = 5;
```

1.6.2 Arithmetic operators (+,-,*,/,%)

+	addition
-	substraction
*	multiplication
/	division
%	modulo

Modulo is the operation that gives the remainder of a division of two values.

1.6.3 Increase an decrease (++,–)

The increase operator (++) and the decrease operator (–) increase or reduce by one the value stored in a variable.Thus:

```
c++;
c+=1;
c=c+1;
```

are all equivalent.
A characteristic of this operator is that it can be used both as a prefix (++a) and as a suffix(a++).Notice the difference :

```
// Case 1
b =3 ;
a = ++b;// a contains 4, b contains 4
// Case 2
b =3 ;
a = b ++;// acontains 3, b contais 4
```

1.6.4 Relational and equality operators (==, !=,>, <, >=, <=)

In order to evaluate a comparison between two expressions we can use the relational and equality operators.

==	Equal to
!=	Note equal to
>	Greater than
<	Less than or equal to
<=	Less than or equal to

1.6.5 Logical operators (!,&&,||)

! operator

The Operator ! is the C++ operator to perform the Boolean operation NOT.

```
!true // evaluates to false
!false // evaluates to true.
```

&& operator

a	b	a&&b
true	true	true
true	false	false
false	true	false
false	false	false

|| operator

a	b	a||b
true	true	true
true	false	true
false	true	true
false	false	false

Conditional operator (?)

The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:

```
condition ? result1 : result2
```

If condition is true the expression will return result1, if it is not it will return result2.

1.6.6 Comma operator (,)

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected.

```
a = (b=3, b+2);
```

Would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

1.6.7 Bitwise operators

&	AND	Bitwise AND
—	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
∼	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

1.6.8 Explicit type casting oerator

Type casting operators allow you to convert a datum of a given type to another.

```
int i;
float f = 3.14;
i = (int) f;
// OR
i = int ( f );
```

1.6.9 sizeof()

This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

```
a = sizeof (char);
```

1.6.10 Precedence of operators

Level	Operator	Description	Grouping
1	::	scope	Left-to-right
2	() [] . − > ++ −	postfix	Left-to-right
3	++ − ∼ sizeof new delete * & +-	unary (prefix) indirection and reference (pointers) unary sign operator	Right-to-left
4	(type)	type casting	Right-to-left
5	.* − >*	pointer-to-member	Left-to-right
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right
8	<<>>	shift	Left-to-right
9	<><=>=	relational	Left-to-right
10	==! =	equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	ˆ	bitwise XOR	Left-to-right
13	|	bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15	||	logical OR	Left-to-right
16	?:	conditional	Right-to-left
17	= *= /= %= += -= >>=<<= & =ˆ= | =	assignemnt	Right-to-left
18	,	comma	Left-to-right

All these precedence levels for operators can be manipulated or become more legible by removing possible ambiguities using parentheses ()

1.7 Basic Input/Output

1.7.1 Standard Output (cout)

By default, the standard output of a program is the screen, and the C++ stream object defined to access it is cout. cout is used in conjunction with the insertion operator <<

```
cout << "Output sentence"; // prints Output sentence on screen
cout << 120; // prints number 120 on screen
cout << x; // prints the content of x on screen
```

The << operator inserts the data that follows it into the stream preceding it.
The insertion operator («) may be used more than once in a single statement:

```
cout << "Hello, I am " << age << " years old and my zipcode is " << zipcode;
```

In C++ a new-line character can be specified as \n
Additionally, to add a new-line, you may also use the endl manipulator.

```
cout << "First sentence." << endl;
```

1.7.2 Standard Input (cin)

The standard input device is usually the keyboard.. Handling the standard input in C++ is done by applying the overloaded operator of extraction (») on the cin stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream.

```
int age;
cin >> age;
```

You can also use cin to request more than one datum input from the user:

```
cin >> a >> b;
// same as
cin >> a;
cin >> b;
```

cin and strings

We can use cin to get strings.However,cin extraction stops reading as soon as if finds any blank space character.
In order to get entire lines, we can use the function getline, which is the more recommendable way to get user input with cin:

```
getline (cin, mystr)
```

The standard header file ¡sstream¿ defines a class called stringstream that allows a string-based object to be treated as a stream.This way we can perform extraction or insertion operations from/to strings,For example, if we want to extract an integer from a string we can write:

```
string mystr ("1204");
int myint;
stringstream(mystr) >> myint;
```

Chapter 2

Control Structures

2.1 Control Structures

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions.

2.1.1 block {}

A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: :

```
{statement1;statement2; statement3;}
```

2.1.2 Conditional structure:if and else

The if keyword is used to execute a statement or block only if a condition is fulfilled.

```
if(condition) statement
```

Where condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored.
We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword else.

```
if (condition) statement1 else statement2
```

The if + else structures can be concatenated with the intention of verifying a range of values.

```
if (x > 0)
 cout << "x is positive";
else if (x < 0)
 cout << "x is negative";
else
 cout << "x is 0";
```

2.1.3 Iteration structures (loops)

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

The while loop

format :

```
while (expression) statement
```

And its functionality is simply to repeat statement while the condition set in expression is true.

The do-while loop

format :

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

Its functionality is exactly the same as the while loop, except that condition in the do-while loop is evaluated after the execution of statement instead of before.

The for loop

format :

```
for (initialization; condition; increase) statement;
```

And its main function is to repeat statement while condition remains true, like the while loop. But in addition, the for loop provides specific locations to contain an initialization statement and an increase statement.
Note that

- The initialization and increase fields are optional.
- Optionally, using the comma operator (,) we can specify more than one expression in any of the fields included in a for loop

2.1.4 Jump statements

The break statement

Using break we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end.

```
break;
```

The continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration.

```
continue;
```

The goto statement

goto allows to make an absolute jump to another point in the program.The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:).
example :

```
include <iostream>
using namespace std;
int main ()
{
 int n=10;
 loop:
 cout << n << ", ";
 n--;
 if (n>0) goto loop;
 cout << "FIRE!\n";
 return 0;
}
```

The exit function

The purpose of exit is to terminate the current program with a specific exit code.

```
    void exit (int exitcode);
```

2.1.5 The selective structure: switch

Its objective is to check several possible constant values for an expression.

```
switch (expression)
{
 case constant1:
 group of statements 1;
 break;
 case constant2:
 group of statements 2;
 break;
 .
 .
 .
 default:
 default group of statements
}
```

- switch evaluates expression and checks if it is equivalent to constant1, if it is, it executes group of statements 1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure.
- If expression was not equal to constant1 it will be checked against constant2. If it is equal to this, it will execute group of statements 2 until a break keyword is found, and then will jump to the end of the switch selective structure.
- Finally, if the value of expression did not match any of the previously specified constants the program will execute the statements included after the default: label, if it exists

2.2 Function

Using functions we can structure our programs in a more modular way.
A function is a group of statements that is executed when it is called from some point of the program.

```
type name ( parameter1, parameter2, ...) { statements }
```

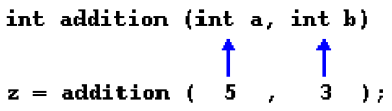
- type is the data type specifier of the data returned by the function, if we want to return no value we use the void type specifier.
- name is the identifier by which it will be possible to call the function.
- parameters :Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- statements is the function's body. It is a block of statements surrounded by braces { }.

the format for calling a function includes specifying its name and enclosing its parameters between parentheses.

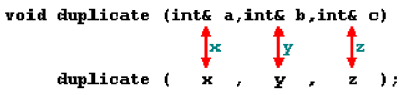
```
printmessage();
```

2.2.1 Arguments passed by value and by reference

- Passing by value :
  This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves.



- Passing by reference:
  When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.
  to pass by reference the type of each parameter was followed by an ampersand sign (&)



2.2.2 Default values in parameters

When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

```
int divide (int a, int b=2){
    ...
}
```

2.2.3 Overloaded functions

In C++ two different functions can have the same name if their parameter types or number are different.

```
#include <iostream>
using namespace std;
int operate (int a, int b)
{
 return (a*b);
}
float operate (float a, float b)
{
 return (a/b);
}
int main ()
{
 int x=5,y=2;
 float n=5.0,m=2.0;
 cout << operate (x,y);
 cout << "\n";
 cout << operate (n,m);
 cout << "\n";
 return 0;
}
```

2.2.4 Recursivity

Recursivity is the property that functions have to be called by themselves.

```
long factorial (long a)
{
 if (a > 1)
 return (a * factorial (a-1));
 else
 return (1);
}
```

2.2.5 Declaring functions

To be able to call a function it must have been declared in some earlier point of the code.
There is an alternative way to avoid writing the whole code of a function before it can be used.
This can be achieved by declaring just a prototype of the function before it is used, instead of the entire definition.

```
type name ( argument_type1, argument_type2, ...);
```

Having the prototype of all functions together in the same place within the source code is found practical by some programmers, and this can be easily achieved by declaring all functions prototypes at the beginning of a program.

Chapter 3

Compound data types

3.1 Arrays

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

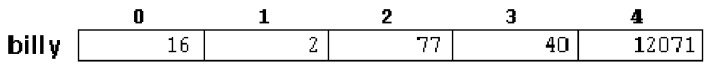
```
type name [elements];
```

where elements field specifies how many of these elements the array has to contain.

3.1.1 Initializing arrays

we have the possibility to assign initial values to each one of its elements by enclosing the values in braces { }

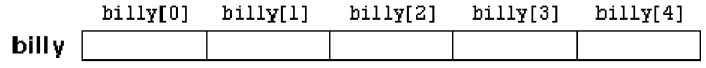
```
int billy [5] = { 16, 2, 77, 40, 12071 };
// OR
int billy [] = { 16, 2, 77, 40, 12071 };
```



3.1.2 Accessing the values of an array

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value.

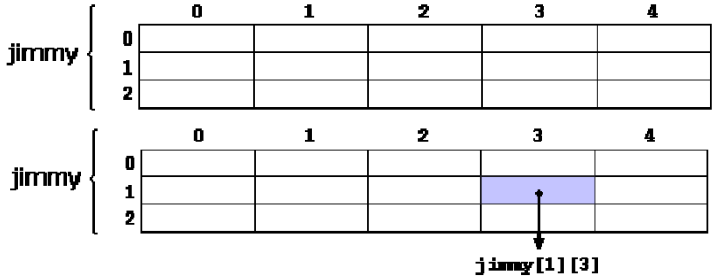
```
name [index]
```




```
billy[2] = 75;// To store a value
a = billy[2]; // to pass the value to a variable
```

3.1.3 Multidimensional arrays

Multidimensional arrays can be described as "arrays of arrays".



declaration:

```
int jimmy [3][5];// declaration
jimmy[1][2];// accessing
```

3.1.4 Arrays as parameters

In C++ it is not possible to pass a complete block of memory by value as a parameter to a function, but we are allowed to pass its address. declaration :

```
void procedure (int arg[])
```

In a function declaration it is also possible to include multidimensional arrays.

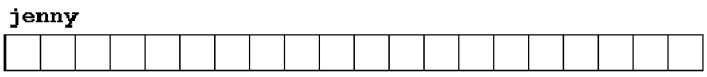
```
base_type[][depth][depth]
```

Notice that the first brackets [] are left blank while the following ones are not. This is so because the compiler must be able to determine within the function which is the depth of each additional dimension.

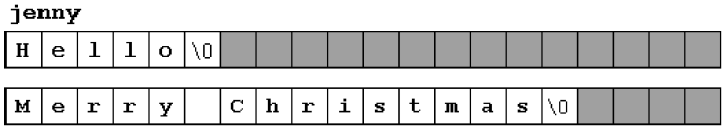
3.2 Character Sequences

because strings are in fact sequences of characters, we can represent them also as plain arrays of char elements.
since the array of characters can store shorter sequences than its total length, a special character is used to signal the end of the valid sequence: the null character, whose literal constant can be written as '\0'

```
char jenny [20];
```



Our array of 20 elements of type char, called jenny, can be represented storing the characters sequences "Hello" and "Merry Christmas" as:



we can initialize the array of char elements called myword with a null-terminated sequence of characters by either one of these two methods:

```
char myword [] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char myword [] = "Hello";
```

sequences of characters stored in char arrays can easily be converted into string objects just by using the assignment operator:

```
string mystring;
char myntcs[]="some text";
mystring = myntcs;
```

3.3 Pointers

The memory of your computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (one byte). These single-byte memory cells are numbered in a consecutive way, so as, within any block of memory, every cell has the same number as the previous one plus one.

3.3.1 Reference operator (&)

The address that locates a variable within memory is what we call a reference to that variable. This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (&)

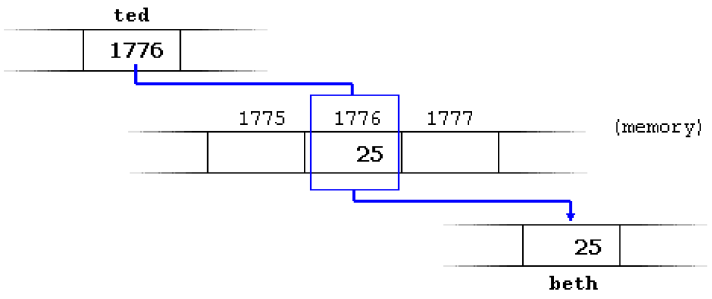
```
ted = &andy;//This would assign to ted the address of variable andy
```

The variable that stores the reference to another variable is what we call a pointer

3.3.2 Dereference operator (*)

Pointers are said to "point to" the variable whose reference they store.
Using a pointer we can directly access the value stored in the variable which it points to. To do this, we simply have to precede the pointer's identifier with an asterisk (*)

```
beth = *ted;// beth equal to value pointed by ted
```



You must clearly differentiate that the expression ted refers to the value 1776, while *ted refers to the value stored at address 1776

```
beth = ted; // beth equal to ted ( 1776 )
beth = *ted; // beth equal to value pointed by ted ( 25 )
```

Notice the difference between the reference and dereference operators:

- & is the reference operator and can be read as "address of"

- * is the dereferenece operator and can be read as "value pointed by"

Thus, they have complementary (or opposite) meanings. A variable referenced with & can be dereferenced with *.

```
andy = 25;
ted = &andy;
// The following expressions are true
andy == 25
&andy == 1776
ted == 1776
*ted == 25
//as long as the address pointed by ted remains unchanged the following expression will be true:
*ted == andy
```

3.3.3 Declaring variables of pointer types

Due to the ability of a pointer to directly refer to the value that it points to, it becomes necessary to specify in its declaration which data type a pointer is going to point to.
The declaraton of pointers :

```
type * name;
```

where type is the data type of the value that the pointer is intended to point to. This type is not the type of the pointer itself! but the type of the data the pointer points to.
the asterisk sign (*) that we use when declaring a pointer only means that it is a pointer, and should not be confused with the dereference operator.
Declaring multiple pointers

```
int * p1, * p2;
```

3.3.4 Pointers and arrays

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the first element that it points to, so in fact they are the same concept.

```
int numbers [20];
int * p;
// the following assignment operation would be valid
p = numbers;
```

After that, p and numbers would be equivalent and would have the same properties. The only difference is that we could change the value of pointer p by another one, whereas numbers will always point to the first of the 20 elements of type int with which it was defined.
Therefore, unlike p, which is an ordinary pointer, numbers is an array, and an array can be considered a constant pointer.Therefore, the following allocation would not be valid:

```
numbers = p;
```

bracket sign operators [] are also a dereference operator known as offset operator. They dereference the variable they follow just as * does, but they also add the number between brackets to the address being dereferenced.

```
//These two expressions are equivalent and valid both if a is a pointer or if a is an array.
a[5] = 0; // a [offset of 5] = 0
*(a+5) = 0; // pointed by (a+5) = 0
```

3.3.5 Pointer intialization

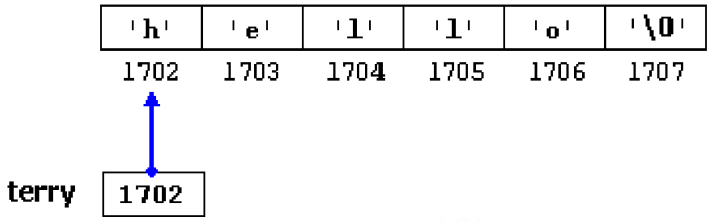
When declaring pointers we may want to explicitly specify which variable we want them to point to:

```
int number;
int *tommy = &number;
// The behavior of this code is equivalent to
int number;
int *tommy;
tommy = &number;
```

When a pointer initialization takes place we are always assigning the reference value to where the pointer points (tommy), never the value being pointed (*tommy).
As in the case of arrays, the compiler allows the special case that we want to initialize the content at which the pointer points with constants at the same moment the pointer is declared:

```
char * terry = "hello";
```

In this case, memory space is reserved to contain ”hello” and then a pointer to the first character of this memory block is assigned to terry.



It is important to indicate that terry contains the value 1702, and not ’h’ nor ”hello”

The pointer terry points to a sequence of characters and can be read as if it was an array,we can access the fifth element of the array with any of these two expression:

```
*(terry+4)
terry[4]
```

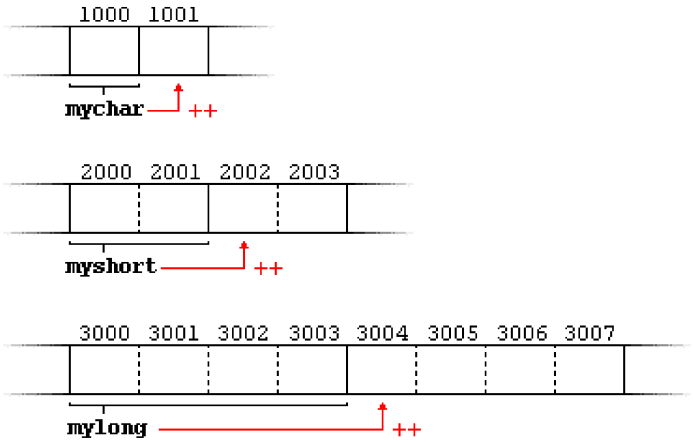
3.3.6 Pointer arithmetics

To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer data types. To begin with, only addition and subtraction operations are allowed to be conducted with them.
But both addition and subtraction have a different behavior with pointers according to the size of the data type to which they point.

let’s assume that in a given compiler for a specific machine, char takes 1 byte, short takes 2 bytes and long takes 4.

```
char *mychar;  // point to 1000
short *myshort;// point to 2000
long *mylong; // point to 3000

mychar++; // point to 1001
myshort++;// point to 2002
mylong++; // point to 3004
//It would happen exactly the same if we write:
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```



Both the increase (++) and decrease (–) operators have greater operator precedence than the dereference operator (*).
Therefore, the following expression may lead to confusion:

```
// those expression are equivalent
*p++ ;
*(p++);
```

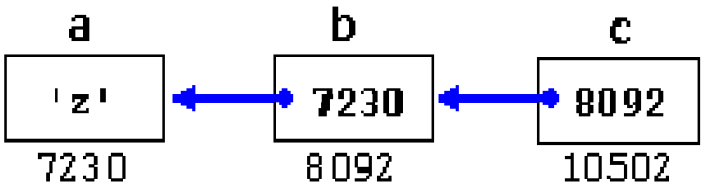
If we write :

```
*p++ = *q++;
// Because ++ has a higher precedence than * It would be roughly equivalent to:
*p = *q;
++p;
++q;
```

3.3.7 Pointers to pointers

C++ allows the use of pointers that point to pointers,we only need to add an asterisk (*) for each level of reference in their declarations:

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```



The value of each variable is written inside each cell; under the cells are their respective addresses in memory.

- c has type char** and a value of 8092
- *c has type char* and a value of 7230
- **c has type char and a value of 'z'

3.3.8 Void pointers

The void type of pointer is a special type of pointer. In C++, void represents the absence of type, so void pointers are pointers that point to a value that has no type.
This allows void pointers to point to any data type, in exchange he data pointed by them cannot be directly dereferenced.
One of its uses may be to pass generic parameters to a function

3.3.9 Null pointer

A null pointer is a value that any pointer may take to represent that it is pointing to ”nowhere”.

```
int * p;
p = 0; // p has a null pointer value
```

3.3.10 Pointers to functions

The typical use of this is for passing a function as an argument to another function, since these cannot be passed dereferenced.
In order to declare a pointer to a function we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name:

```
#include <iostream>
using namespace std;
int addition (int a, int b) { return (a+b); }
int subtraction (int a, int b) { return (a-b); }
int operation (int x, int y, int(*functocall)(int,int)) {
int g;
g = (*functocall)(x,y);
return (g);
}
int main () {
int m,n;
int (*minus)(int,int) = subtraction; //minus is a pointer to a function that has two parameters
m = operation (7, 5, addition);
n = operation (20, m, minus);
cout <<n;
return 0;
}

Output :
    8
```

3.4 Dynamic memory

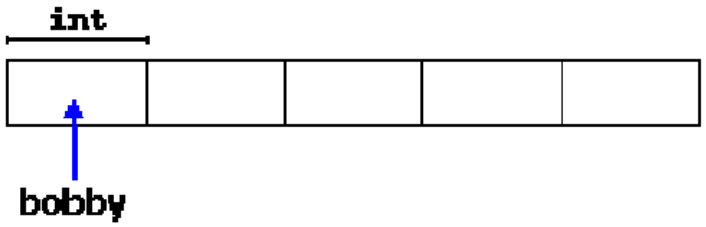
if we need a variable amount of memory that can only be determined during runtime we must use dynamic memory.

3.4.1 Operators new and new[]

In order to request dynamic memory we use the operator new followed by a data type specifier.
It returns a pointer to the beginning of the new block of memory allocated.

```
pointer = new type //allocate memory to contain one single element
pointer = new type [number_of_elements] //assign a block (an array) of elements of type type,
// exemple
int * bobby;
bobby = new int [5];
```

The system dynamically assigns space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to bobby.
Therefore, now, bobby points to a valid block of memory with space for five elements of type int.



Difference between declaring a normal array and assigning dynamic memory to a pointer

the dynamic memory allocation allows us to assign memory during the execution of the program (runtime) using any variable or constant value as its size.

3.4.2 Check if the allocation was successful

Computer memory is a limited resource, when a memory allocation fails, terminating the program, the pointer returned by new is a null pointer.

```
int * bobby;
bobby = new (nothrow) int [5];
if (bobby == 0) {
 // error assigning memory. Take measures.
 };
```

3.4.3 Operators delete and delete[]

Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again.

```
delete pointer; //  delete memory allocated for a single element
delete [] pointer;// delete memory allocated for arrays of elements.
```

The value passed as argument to delete must be a pointer to a memory block previously allocated with new.

3.5 Data Structure

A data structure is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different lengths.

```
struct structure_name {
    member_type1 member_name1;
    member_type2 member_name2;
    member_type3 member_name3;
    .
    .
} object_names;
```

- structure_name is a name for the structure type
- Right at the end of the struct declaration, and before the ending semicolon, we can use the optional field object_name to directly declare objects of the structure type.

we have to know is that a data structure creates a new type: Once a data structure is declared, a new type with the identifier specified as structure_name is created and can be used in the rest of the program

```
struct product {
    int weight;
    float price;
} ;
product apple;
product banana, melon;
```

We can use a dot (.) to operate directly with the objects member as if they were standard variables.

```
apple.weight
apple.price
banana.weight
banana.price
melon.weight
melon.price
```


3.5.1 Pointer to structures

Like any other type, structures can be pointed by its own type of pointers.
The arrow operator $(->)$ is a dereference operator that is used exclusively with pointers to objects with members.

```
struct movies_t {
 string title;
 int year;
};
movies_t amovie;
movies_t * pmovie;
// the following code would also be valid
pmovie = &amovie;
// The following code is equivalent to (*pmovie).title
pmovie->year;
// note that  *pmovie.title is equivalent to *(pmovie.title)
```

3.5.2 Nesting structures

Structures can also be nested so that a valid element of a structure can also be in its turn another structure.

```
struct movies_t {
    string title;
    int year;
};
struct friends_t {
    string name;
    string email;
    movies_t favorite_movie;
 } charlie, maria;
friends_t * pfriends = &charlie;
// we could use any of the following expressions :
charlie.name
maria.favorite_movie.title
// the two following expressions are the same member
charlie.favorite_movie.year
pfriends->favorite_movie.year
```

Chapter 4

Object Oriented Programming

4.1 Classes

A class is an expanded concept of a data structure:it can hold both data and functions.
An object is an instantiation of a class.

```
class class_name {
    access_specifier_1:
    member1;
    access_specifier_2:
    member2;
    ...
} object_names;
```

The body of the declaration can contain members, that can be either data or function declarations,and optionally access specifiers.

An access specifier is one of the following three keywords: private, public or protected. These specifiers modify the access rights that the members following them acquire:

- private (accessible from members of the same class), default access.
- protected (accessible form members from the same class and from their derived classes)
- public (accessible from anywhere)

```
class CRectangle {
    int x, y;
    public:
    void set_values (int,int);
    int area (void);
 } rect;
```

We can to any of the public members of the object as if they were normal functions or normal variables, just by putting the object's name followed by a dot (.) and then the name of the member.

```
    rect.set_values (3,4);
    myarea = rect.area();
```

The two colons "::" is used to define a member of a class from outside the calss definition.

```
void CRectangle::set_values (int a, int b) {
 x = a;
 y = b;
}
```

That is the basic concept of object-oriented programming: Data and functions are both members of the object.

4.1.1 Constructors and destructors

Constructors

A class can include a special function called constructor, which is automatically called whenever a new object of this class is created.
This constructor function must have the same name as the class, and cannot have any return type; not even void.

```
#include <iostream>
using namespace std;
class CRectangle {
 int width, height;
 public:
 CRectangle (int,int);
 int area () {return (width*height);}
};
CRectangle::CRectangle (int a, int b) {
 width = a;
 height = b;
}
int main () {
 CRectangle rect (3,4);
 CRectangle rectb (5,6);
 cout << "rect area: " << rect.area() << endl;
 cout << "rectb area: " << rectb.area() << endl;
 return 0;
}
-----------------
Output:
rect area: 12
rectb area: 30
```

Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created.

Overloading

A constructor can also be overloaded with more than one function that have the same name but different types or number of parameters.

```
#include <iostream>
using namespace std;
class CRectangle {
 int width, height;
```

```
 public:
 CRectangle ();
 CRectangle (int,int);
 int area (void) {return (width*height);}
};
CRectangle::CRectangle () {
 width = 5;
 height = 5;
}
CRectangle::CRectangle (int a, int b) {
 width = a;
 height = b;
}
int main () {
 CRectangle rect (3,4);
 CRectangle rectb;
 cout << "rect area: " << rect.area() << endl;
 cout << "rectb area: " << rectb.area() << endl;
 return 0;
}
-------------
Output:
rect area: 12
rectb area: 25
```

Important : Notice how if we declare a new object and we want to use its default constructor (the one without parameters), we do not include parentheses ():

```
CRectangle rectb; // right
CRectangle rectb(); // wrong!
```

Default constructor

The compiler creates a default constructor for you if you do not specify your own. It provides three special member functions in total that are implicitly declared

- the copy constructor
- the copy assignemnt operator
- the default destructor

The copy constructor and the copy assignment operator copy all the data contained in another object to the data members of the current object.
it would be somthing like

```
Classname::Classname (const Classname& rv) {
 a=rv.a; b=rv.b; c=rv.c;
 }
```

Therefore

```
// using the constructor maded by the programmers
Classname object1 (2,3);
// using the default constructo
Classname object2 (object1);// we copy the object1
```

Destructor

The destructor fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished or because it is an object dynamically assigned and it is released using the operator delete.
The destructor must have the same name as the class, but preceded with a tilde sign (∼) and it must also return no value.

The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

```
#include <iostream>
using namespace std;
class CRectangle {
 int *width, *height;
 public:
 CRectangle (int,int);
 ~CRectangle ();
 int area () {return (*width * *height);}
};
CRectangle::CRectangle (int a, int b) {
 width = new int;
 height = new int;
 *width = a;
 *height = b;
}
CRectangle::~CRectangle () {
 delete width;
 delete height;
}
```


4.1.2 Pointers to classes

A class becomes a valid type, so we can use the class name as the type for the pointer.
As it happened with data structures, in order to refer directly to a member of an object pointed by a pointer we can use the arrow operator ($->$) of indirection.

4.1.3 Overloading operators

When you define a class, you are actually creating a new type.
Most of the C++ operators can be overloaded to apply to your new class type.

Overloadable operators												
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	|
~	&=	^=	|=	&&	||	%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

To overload an operator in order to use it with classes we declare operator functions, which are regular functions whose names are the operator keyword followed by the operator sign that we want to overload.

```
type operator sign (parameters) { /*...*/ }
```

Example :

```
// vectors: overloading operators example
#include <iostream>
using namespace std;
class CVector {
 public:
 int x,y;
 CVector () {};
 CVector (int,int);
 CVector operator + (CVector);
};
CVector::CVector (int a, int b) {
 x = a;
 y = b;
}
CVector CVector::operator+ (CVector param) {
 CVector temp;
 temp.x = x + param.x;
 temp.y = y + param.y;
 return (temp);
}
int main () {
 CVector a (3,1);
 CVector b (1,2);
 CVector c;
 c = a + b;
 cout << c.x << "," << c.y;
 return 0;
}
--------------
Output :
4,3
```


Both expressions are equivalent :

```
c = a + b;
c = a.operator+ (b);
```

4.1.4 The keyword this

The keyword this represents a pointer to the object whose member function is being executed. It is a pointer to the object itself.
Example :

```
#include <iostream>
using namespace std;
class CDummy {
 public:
 int isitme (CDummy& param);
};
int CDummy::isitme (CDummy& param)
{
 if (&param == this) return true;
 else return false;
}
```

```
int main () {
 CDummy a;
 CDummy* b = &a;
 if ( b->isitme(a) )
 cout << "yes, &a is b";
 return 0;
}
```

4.1.5 Static members

Static data members of a class are also known as "class variables", because there is only one unique value for all the objects of that same class. Their content is not different from one object of this class to another. it may be used for a variable within a class that can contain a counter with the number of objects of that class that are currently allocated

```
#include <iostream>
using namespace std;
class CDummy {
 public:
 static int n;
 CDummy () { n++; };
 ~CDummy () { n--; };
};
int CDummy::n=0;
int main () {
 CDummy a;
 CDummy b[5];
 CDummy * c = new CDummy;
 cout << a.n << endl;
 delete c;
 cout << CDummy::n << endl;
 return 0;
}
```


In fact, static members have the same properties as global variables but they enjoy class scope.
we can only include the prototype (its declaration) in the class declaration but not its definition (its initialization).
In order to initialize a static data-member we must include a formal definition outside the class.

Chapter 5

Input/Output with files

C++ provides the following classes to perform output and input of characters to/from files:

- ofstream: Stream class to write on files.
- ifstream: Stream class to read from files.
- fstream: Stream class to both read and write from/to files.

5.1 Open a file

In order to open a file with a stream object we use its member function open():

```
open (filename,mode);
```

Where filename representing the name of the file to be opened and mode is an optional parameter with a combination of the following flags:

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations.
ios::trunc	If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.

All these flags can be combined using the bitwise operator OR (—). Each one of the open() member functions of the classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

Note that : File streams opened in binary mode perform input and output operations independently of any format considerations.

To check if a file stream was successful opening a file, you can do it by calling to member is_open()

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```

5.2 Closing a file

When we are finished with our input and output operations on a file we shall close it so that its resources become available again.
In order to do that we have to call the stream's member function close().

```
myfile.close();
```

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function close().

5.3 Text files

Text file streams are those where we do not include the ios::binary flag in their opening mode.

5.3.1 Writing on a text file

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
 ofstream myfile ("example.txt");
 if (myfile.is_open())
 {
 myfile << "This is a line.\n";
 myfile << "This is another line.\n";
 myfile.close();
 }
 else cout << "Unable to open file";
 return 0;
}
```

5.3.2 reading a text file

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main () {
 string line;
 ifstream myfile ("example.txt");
 if (myfile.is_open())
 {
 while (! myfile.eof() )
 {
 getline (myfile,line);
 cout << line << endl;
 }
 myfile.close();
 }
 else cout << "Unable to open file";
 return 0;
}
```

The function eof() returns true in the case that the end of the file has been reached.