First week

Structure of a program

The first panel shows the source code for our first program. The second one shows the result of the program once compiled and executed. The way to edit and compile a program depends on the compiler you are using. Depending on whether it has a Development Interface or not and on its version. Consult the compilers section and the manual or help included with your compiler if you have doubts on how to compile a C++ console program.

The previous program is the typical program that programmer apprentices write for the first time, and its result is the printing on screen of the "Hello World!" sentence. It is one of the simplest programs that can be written in C++, but it already contains the fundamental components that every C++ program has. We are going to look line by line at the code we have just written:

```
// my first program in C++

#include <iostream>
using namespace std;

int main ()
{
  cout << "Hello World!";
  return 0;
}
```

```
Hello World!
```

// my first program in C++

This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not have any effect on the behavior of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

#include <iostream>

Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case the directive #include <iostream> tells the preprocessor to include the iostream standard file. This specific file (iostream) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

using namespace std;

All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name std. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library, and in fact it will be included in most of the source codes included in these tutorials.

int main ()

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it - the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function.

The word main is followed in the code by a pair of parentheses (()). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions are these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them.

Right after these parentheses we can find the body of the main function enclosed in braces ({}). What is contained within these braces is what the function does when it is executed.

cout << "Hello World!";

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program.

cout represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is the screen).

cout is declared in the iostream standard file within the std namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code.

Notice that the statement ends with a semicolon character (;). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).

return 0;

The return statement causes the main function to finish. return may be followed by a return code (in our example is followed by the return code 0). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

You may have noticed that not all the lines of this program perform actions when the code is executed. There were lines containing only comments (those beginning by //). There were lines with directives for the compiler's preprocessor (those beginning by #). Then there were lines that began the declaration of a function (in this case, the main function) and, finally lines with statements (like the insertion into cout), which were all included within the block delimited by the braces ({}) of the main function.

The program has been structured in different lines in order to be more readable, but in C++, we do not have strict rules on how to separate instructions in different lines.

Comments

Comments are parts of the source code disregarded by the compiler. They simply do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code.

C++ supports two ways to insert comments:

// line comment

/* block comment */

The first of them, known as line comment, discards everything from where the pair of slash signs (//) is found up to the end of that same line. The second one, known as block comment, discards everything between the /* characters and the first appearance of the */ characters, with the possibility of including more than one line.

We are going to add comments to our second program:

```
/* my second program in C++
   with more comments */
#include <iostream>
```

```
using namespace std;


int main ()

{

 cout << "Hello World! ";     // prints Hello World!

 cout << "I'm a C++ program"; // prints I'm a C++ program

 return 0;

}
```

```
Hello World! I'm a C++ program
```

If you include comments within the source code of your programs without using the comment characters combinations //, /* or */, the compiler will take them as if they were C++ expressions, most likely causing one or several error messages when you compile it.

Q1 Write a program in C++ to print number from (1-4) in horizontal line?

Q2 Write a program in C++ to find number 7 from two other numbers and print X *Y =12? H.W

Q3 Write a program in C++ to find number 8 from two other numbers?

Second week

Variables Data Types.

The usefulness of the "Hello World" programs shown in the previous section is quite questionable. We had to write several lines of code, compile them, and then execute the resulting program just to obtain a simple sentence written on the screen as result. It certainly would have been much faster to type the output sentence by ourselves. However, programming is not limited only to printing simple texts on the screen. In order to go a little further on and to become able to write programs that perform useful tasks that really save us work we need to introduce the concept of variable.

Let us think that I ask you to retain the number 5 in your mental memory, and then I ask you to memorize also the number 2 at the same time. You have just stored two different values in your memory. Now, if I ask you to add 1 to the first number I said, you should be retaining the numbers 6 (that is 5+1) and 2 in your memory. Values that we could now for example subtract and obtain 4 as result.

The whole process that you have just done with your mental memory is a simile of what a computer can do with two variables. The same process can be expressed in C++ with the following instruction set:

```
a = 5;

b = 2;

a = a + 1;

result = a - b;
```

Obviously, this is a very simple example since we have only used two small integer values, but consider that your computer can store millions of numbers like these at the same time and conduct sophisticated mathematical operations with them.

Therefore, we can define a variable as a portion of memory to store a determined value.

Each variable needs an identifier that distinguishes it from the others, for example, in the previous code the variable identifiers were a, b and result, but we could have called the variables any names we wanted to invent, as long as they were valid identifiers.

Identifiers

A valid identifier is a sequence of one or more letters, digits or underscores characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid. In addition, variable identifiers always have to begin with a letter. They can also begin with an underline character (_), but in some cases these may be reserved for compiler specific keywords or external identifiers,

as well as identifiers containing two successive underscore characters anywhere. In no case they can begin with a digit.

Another rule that you have to consider when inventing your own identifiers is that they cannot match any keyword of the C++ language nor your compiler's specific ones, which are *reserved keywords*. The standard reserved keywords are:

```
asm, auto, bool, break, case, catch, char, class, const, const_cast,
continue, default, delete, do, double, dynamic_cast, else, enum,
explicit, export, extern, false, float, for, friend, goto, if,
inline, int, long, mutable, namespace, new, operator, private,
protected, public, register, reinterpret_cast, return, short, signed,
sizeof, static, static_cast, struct, switch, template, this, throw,
true, try, typedef, typeid, typename, union, unsigned, using,
virtual, void, volatile, wchar_t, while
```

Additionally, alternative representations for some operators cannot be used as identifiers since they are reserved words under some circumstances:

```
and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor,
xor_eq
```

Your compiler may also include some additional specific reserved keywords.

Very important: The C++ language is a "case sensitive" language. That means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the `RESULT` variable is not the same as the `result` variable or the `Result` variable. These are three different variable identifiers.

Fundamental data types

When programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

The memory in our computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between 0 and 255). In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

Name	**Description**	**Size***	**Range***
`char`	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
`short int` (`short`)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
`int`	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295

long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character.	2 *or* 4 bytes	1 wide character

* The values of the columns **Size** and **Range** depend on the system the program is compiled for. The values shown above are those found on most 32-bit systems. But for other systems, the general specification is that `int` has the natural size suggested by the system architecture (one "*word*") and the four integer types `char`, `short`, `int` and `long` must each one be at least as large as the one preceding it, with `char` being always 1 byte in size. The same applies to the floating point types `float`, `double` and `long double`, where each one must provide at least as much precision as the preceding one.

Declaration of variables

In order to use a variable in C++, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float...) followed by a valid variable identifier. For example:

```
int a;
float mynumber;
```

These are two valid declarations of variables. The first one declares a variable of type `int` with the identifier `a`. The second one declares a variable of type `float` with the identifier `mynumber`. Once declared, the variables `a` and `mynumber` can be used within the rest of their scope in the program.

If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

```
int a, b, c;
```

This declares three variables (`a`, `b` and `c`), all of them of type `int`, and has exactly the same meaning as:

```
int a;
int b;
int c;
```

The integer data types `char`, `short`, `long` and `int` can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specifier `signed` or the specifier `unsigned` before the type name. For example:

```
unsigned short int NumberOfSisters;
signed int MyAccountBalance;
```

By default, if we do not specify either `signed` or `unsigned` most compiler settings will assume the type to be signed, therefore instead of the second declaration above we could have written:

```
int MyAccountBalance;
```

with exactly the same meaning (with or without the keyword `signed`)

An exception to this general rule is the `char` type, which exists by itself and is considered a different fundamental data type from `signed char` and `unsigned char`, thought to store characters. You should use either `signed` or `unsigned` if you intend to store numerical values in a `char`-sized variable.

`short` and `long` can be used alone as type specifiers. In this case, they refer to their respective integer fundamental types: `short` is equivalent to `short int` and `long` is equivalent to `long int`. The following two variable declarations are equivalent:

```
short Year;
short int Year;
```

Finally, `signed` and `unsigned` may also be used as standalone type specifiers, meaning the same as `signed int` and `unsigned int` respectively. The following two declarations are equivalent:

```
unsigned NextYear;
unsigned int NextYear;
```

To see what variable declarations look like in action within a program, we are going to see the C++ code of the example about your mental memory proposed at the beginning of this section:

```
// operating with variables

#include <iostream>
using namespace std;

int main ()
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  a = 5;
  b = 2;
  a = a + 1;
```

```
4
```

```
  result = a - b;

  // print out the result:
  cout << result;

  // terminate the program:
  return 0;
}
```

Do not worry if something else than the variable declarations themselves looks a bit strange to you. You will see the rest in detail in coming sections.

Constants

Constants are expressions with a fixed value.

Literals

Literals are used to express particular values within the source code of a program. We have already used these previously to give concrete values to variables or to express messages we wanted our programs to print out, for example, when we wrote:

```
a = 5;
```

the `5` in this piece of code was a literal constant.

Literal constants can be divided in Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Integer Numerals

```
1776
707
-273
```

They are numerical constants that identify integer decimal values. Notice that to express a numerical constant we do not have to write quotes (") nor any special character. There is no doubt that it is a constant: whenever we write `1776` in a program, we will be referring to the value 1776.

In addition to decimal numbers (those that all of us are used to use every day) C++ allows the use as literal constants of octal numbers (base 8) and hexadecimal numbers (base 16). If we want to express an octal number we have to precede it with a `0` (zero character). And in order to express a hexadecimal number we have to precede it with the characters `0x` (zero, x). For example, the following literal constants are all equivalent to each other:

```
75          // decimal
0113        // octal
0x4b        // hexadecimal
```

All of these represent the same number: 75 (seventy-five) expressed as a base-10 numeral, octal numeral and hexadecimal numeral, respectively.

Literal constants, like variables, are considered to have a specific data type. By default, integer literals are of type `int`. However, we can force them to either be unsigned by appending the `u` character to it, or long by appending `l`:

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

In both cases, the suffix can be specified using either upper or lowercase letters.

Floating Point Numbers

They express numbers with decimals and/or exponents. They can include either a decimal point, an `e` character (that expresses "by ten at the Xth height", where X is an integer value that follows the `e` character), or both a decimal point and an `e` character:

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

These are four valid numbers with decimals expressed in C++. The first number is PI, the second one is the number of Avogadro, the third is the electric charge of an electron (an extremely small number) -all of them approximated- and the last one is the number three expressed as a floating-point numeric literal.

The default type for floating point literals is `double`. If you explicitly want to express a `float` or `long double` numerical literal, you can use the `f` or `l` suffixes respectively:

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

Any of the letters than can be part of a floating-point numerical constant (`e`, `f`, `l`) can be written using either lower or uppercase letters without any difference in their meanings.

Q1 Write a program in C++ to find the summation of two number s float?

Q2 Write a program in C++ to find the result of (x-b+a)?

Third week

Character and string literals

There also exist non-numerical constants, like:

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

The first two expressions represent single character constants, and the following two represent string literals composed of several characters. Notice that to represent a single character we enclose it between single quotes (') and to express a string (which generally consists of more than one character) we enclose it between double quotes (").

When writing both single character and string literals, it is necessary to put the quotation marks surrounding them to distinguish them from possible variable identifiers or reserved keywords. Notice the difference between these two expressions:

```
x
'x'
```

`x` alone would refer to a variable whose identifier is `x`, whereas `'x'` (enclosed within single quotation marks) would refer to the character constant `'x'`.

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, like newline (`\n`) or tab (`\t`). All of them are preceded by a backslash (`\`). Here you have a list of some of such escape codes:

`\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 (\)

For example:

```
'\n'
'\t'
"Left \t Right"
"one\ntwo\nthree"
```

Additionally, you can express any character by its numerical ASCII code by writing a backslash character (\) followed by the ASCII code expressed as an octal (base-8) or hexadecimal (base-16) number. In the first case (octal) the digits must immediately follow the backslash (for example \23 or \40), in the second case (hexadecimal), an x character must be written before the digits themselves (for example \x20 or \x4A).

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:

```
"this forms" "a single" "string" "of characters"
```

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"
```

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

Operators

Once we know of the existence of variables and constants, we can begin to operate with them. For that purpose, C++ integrates operators. Unlike other languages whose operators are mainly keywords, operators in C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes C++ code shorter and more international, since it relies less on English words, but requires a little of learning effort in the beginning.

You do not have to memorize all the content of this page. Most details are only provided to serve as a later reference in case you need it.

Assignment (=)

The assignment operator assigns a value to a variable.

```
a = 5;
```

This statement assigns the integer value 5 to the variable a. The part at the left of the assignment operator (=) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value). The lvalue has to be a variable whereas the rvalue can be either a constant, a variable, the result of an operation or any combination of these.
The most important rule when assigning is the *right-to-left* rule: The assignment operation always takes place from right to left, and never the other way:

```
a = b;
```

This statement assigns to variable a (the lvalue) the value contained in variable b (the rvalue). The value that was stored until this moment in a is not considered at all in this operation, and in fact that value is lost.

Consider also that we are only assigning the value of b to a at the moment of the assignment operation. Therefore a later change of b will not affect the new value of a.

For example, let us have a look at the following code - I have included the evolution of the content stored in the variables as comments:

```
// assignment operator

#include <iostream>
using namespace std;

int main ()
{
  int a, b;          // a:?,  b:?
  a = 10;            // a:10, b:?
  b = 4;             // a:10, b:4
  a = b;             // a:4,  b:4
  b = 7;             // a:4,  b:7

  cout << "a:";
  cout << a;
  cout << " b:";
  cout << b;

  return 0;
}
```

```
a:4 b:7
```

This code will give us as result that the value contained in a is 4 and the one contained in b is 7. Notice how a was not affected by the final modification of b, even though we declared a = b earlier (that is because of the *right-to-left rule*).

A property that C++ has over other programming languages is that the assignment operation can be used as the rvalue (or part of an rvalue) for another assignment operation. For example:

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

is equivalent to:

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

that means: first assign 5 to variable b and then assign to a the value 2 plus the result of the previous assignment of b (i.e. 5), leaving a with a final value of 7.

The following expression is also valid in C++:

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

It assigns 5 to the all the three variables: a, b and c.

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

The five arithmetical operations supported by the C++ language are:

+	addition
-	subtraction
*	multiplication
/	division
%	modulo

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see is *modulo*; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:

```
a = 11 % 3;
```

the variable a will contain the value 2, since 2 is the remainder from dividing 11 between 3.

Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators:

expression	**is equivalent to**
`value += increase;`	`value = value + increase;`
`a -= 5;`	`a = a - 5;`
`a /= b;`	`a = a / b;`
`price *= units + 1;`	`price = price * (units + 1);`

and the same for all other operators. For example:

```
// compound assignment operators
```
```
5
```

```
#include <iostream>
using namespace std;

int main ()
{
  int a, b=3;
  a = b;
  a+=2;               // equivalent
to a=a+2
  cout << a;
  return 0;
}
```

Increase and decrease (++, --)

Shortening even more some expressions, the increase operator (`++`) and the decrease operator (`--`) increase or reduce by one the value stored in a variable. They are equivalent to `+=1` and to `-=1`, respectively. Thus:

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

are all equivalent in its functionality: the three of them increase by one the value of c.

In the early C compilers, the three previous expressions probably produced different executable code depending on which one was used. Nowadays, this type of code optimization is generally done automatically by the compiler, thus the three expressions should produce exactly the same executable code.

A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (`++a`) or after it (`a++`). Although in simple expressions like `a++` or `++a` both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix (`a++`) the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression. Notice the difference:

Example 1	**Example 2**

`B=3;`
`A=++B;`
`// A contains 4, B contains 4`
	`B=3;`
	`A=B++;`
	`// A contains 3, B contains 4`

In Example 1, `B` is increased before its value is copied to `A`. While in Example 2, the value of `B` is copied to `A` and then `B` is increased.

Q1 Write a program in C++ to find the difference between
A =++b, A=b++?

Q2 find the result of the following equation (a%b-C*d) in C++ and print the value? H.W

Q3 Write a program in C++ to find the result of (a/b) and (a%b)?

Forth week

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

In order to evaluate a comparison between two expressions we can use the relational and equality operators. The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result.

We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other is. Here is a list of the relational and equality operators that can be used in C++:

==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Here there are some examples:

```
(7 == 5)      // evaluates to false.
(5 > 4)       // evaluates to true.
(3 != 2)      // evaluates to true.
(6 >= 6)      // evaluates to true.
(5 < 5)       // evaluates to false.
```

Of course, instead of using only numeric constants, we can use any valid expression, including variables. Suppose that `a=2`, `b=3` and `c=6`,

```
(a == 5)      // evaluates to false since a is not equal to 5.
(a*b >= c)    // evaluates to true since (2*3 >= 6) is true.
(b+4 > a*c)   // evaluates to false since (3+4 > 2*6) is false.
((b=2) == a) // evaluates to true.
```

Be careful! The operator `=` (one equal sign) is not the same as the operator `==` (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one (`==`) is the equality operator that compares whether both expressions in the two sides of it are equal to each other. Thus, in the last expression `((b=2) == a)`, we first assigned the value `2` to `b` and then we compared it to `a`, that also stores the value `2`, so the result of the operation is true.

Logical operators (!, &&, ||)

The Operator ! is the C++ operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

```
!(5 == 5)     // evaluates to false because the expression at its
right (5 == 5) is true.
!(6 <= 4)     // evaluates to true because (6 <= 4) would be false.
!true         // evaluates to false
!false        // evaluates to true.
```

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The following panel shows the result of operator && evaluating the expression a && b:

&& OPERATOR

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

The operator || corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of a || b:

|| OPERATOR

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

For example:

```
( (5 == 5) && (3 > 6) )  // evaluates to false ( true && false ).
( (5 == 5) || (3 > 6) )  // evaluates to true ( true || 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`.

```
7==5 ? 4 : 3      // returns 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3    // returns 4, since 7 is equal to 5+2.
5>3 ? a : b         // returns the value of a, since 5 is greater
than 3.
a>b ? a : b       // returns whichever is greater, a or b.
```

```
// conditional operator

#include <iostream>
using namespace std;

int main ()
{
  int a,b,c;

  a=2;
  b=7;
  c = (a>b) ? a : b;

  cout << c;

  return 0;
}
```

```
7
```

In this example `a` was `2` and `b` was `7`, so the expression being evaluated (`a>b`) was not true, thus the first value specified after the question mark was discarded in favor of the second value (the one after the colon) which was `b`, with a value of `7`.

Comma operator (,)

The comma operator (`,`) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

For example, the following code:

```
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`.

Bitwise Operators (&, |, ^, ~, <<, >>)

Bitwise operators modify variables considering the bit patterns that represent the values they store.

operator	asm equivalent	description
&	AND	Bitwise AND
|	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

Explicit type casting operator

Type casting operators allow you to convert a datum of a given type to another. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (`()`):

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

The previous code converts the float number `3.14` to an integer value (`3`), the remainder is lost. Here, the typecasting operator was `(int)`. Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int ( f );
```

Both ways of type casting are valid in C++.

Basic Input/Output

Until now, the example programs of previous sections provided very little interaction with the user, if any at all. Using the standard input and output library, we will be able to interact with the user by printing messages on the screen and getting the user's input from the keyboard.

C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen or the keyboard. A stream is an object where a program can either insert or extract characters to/from it. We do not really need to care about many specifications about the physical media associated with the stream - we only need to know it will accept or provide characters sequentially.

The standard C++ library includes the header file `iostream`, where the standard input and output stream objects are declared.

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*, which is written as << (two "less than" signs).

```
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. In the examples above it inserted the constant string Output sentence, the numerical constant 120 and variable x into the standard output stream cout. Notice that the sentence in the first instruction is enclosed between double quotes (") because it is a constant string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes (") so that they can be clearly distinguished from variable names. For example, these two sentences have very different results:

```
cout << "Hello";  // prints Hello
cout << Hello;    // prints the content of Hello variable
```

The insertion operator (<<) may be used more than once in a single statement:

```
cout << "Hello, " << "I am " << "a C++ statement";
```

This last statement would print the message Hello, I am a C++ statement on the screen. The utility of repeating the insertion operator (<<) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

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

If we assume the age variable to contain the value 24 and the zipcode variable to contain 90064 the output of the previous statement would be:

```
Hello, I am 24 years old and my zipcode is 90064
```

It is important to notice that cout does not add a line break after its output unless we explicitly indicate it, therefore, the following statements:

```
cout << "This is a sentence.";
cout << "This is another sentence.";
```

will be shown on the screen one following the other without any line break between them:

```
This is a sentence.This is another sentence.
```

even though we had written them in two different insertions into cout. In order to perform a line break on the output we must explicitly insert a new-line character into cout. In C++ a new-line character can be specified as \n (backslash, n):

```
cout << "First sentence.\n ";
cout << "Second sentence.\nThird sentence.";
```

This produces the following output:

```
First                                                   sentence.
Second                                                  sentence.
Third sentence.
```

Additionally, to add a new-line, you may also use the `endl` manipulator. For example:

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

would print out:

```
First                                                   sentence.
Second sentence.
```

The `endl` manipulator produces a newline character, exactly as the insertion of `'\n'` does, but it also has an additional behavior when it is used with buffered streams: the buffer is flushed. Anyway, `cout` will be an unbuffered stream in most cases, so you can generally use both the `\n` escape character and the `endl` manipulator in order to specify a new line without any difference in its behavior.

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. For example:

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

The first statement declares a variable of type `int` called `age`, and the second one waits for an input from `cin` (the keyboard) in order to store it in this integer variable.

`cin` can only process the input from the keyboard once the `RETURN` key has been pressed. Therefore, even if you request a single character, the extraction from `cin` will not process the input until the user presses `RETURN` after the character has been introduced.

You must always consider the type of the variable that you are using as a container with `cin` extractions. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters.

```
// i/o example

#include <iostream>
using namespace std;

int main ()
{
  int i;
  cout << "Please enter an integer
value: ";
  cin >> i;
  cout << "The value you entered
is " << i;
  cout << " and its double is " <<
i*2 << ".\n";
```

```
Please enter an integer value: 702
The value you entered is 702 and
its double is 1404.
```

```
  return 0;
}
```

The user of a program may be one of the factors that generate errors even in the simplest programs that use `cin` (like the one we have just seen). Since if you request an integer value and the user introduces a name (which generally is a string of characters), the result may cause your program to misoperate since it is not what we were expecting from the user. So when you use the data input provided by `cin` extractions you will have to trust that the user of your program will be cooperative and that he/she will not introduce his/her name or something similar when an integer value is requested. A little ahead, when we see the `stringstream` class we will see a possible solution for the errors that can be caused by this type of user input.

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

```
cin >> a >> b;
```

is equivalent to:

```
cin >> a;
cin >> b;
```

Q1 Write a program in C++ to inter two integer numbers a and b then the program should print the following a+b , a-b , a*b , a/b.

Q2 Write a program in C++ to print each of (integer number ,add (9) to it, multiply the new number by(3) , minus (7) from the new number)

Fifth week

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. For that purpose, C++ provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

With the introduction of control structures we are going to have to introduce a new concept: the *compound-statement* or *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; }
```

Most of the control structures that we will see in this section require a generic statement as part of its syntax. A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compound statement (several instructions grouped in a block), like the one just described. In the case that we want the statement to be a simple statement, we do not need to enclose it in braces (`{}`). But in the case that we want the statement to be a compound statement it must be enclosed between braces (`{}`), forming a block.

Conditional structure: if and else

The `if` keyword is used to execute a statement or block only if a condition is fulfilled. Its form is:

```
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 (not executed) and the program continues right after this conditional structure.
For example, the following code fragment prints `x is 100` only if the value stored in the `x` variable is indeed `100`:

```
if (x == 100)
  cout << "x is 100";
```

If we want more than a single statement to be executed in case that the condition is true we can specify a block using braces `{ }`:

```
if (x == 100)
{
   cout << "x is ";
   cout << x;
}
```

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword `else`. Its form used in conjunction with `if` is:

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

For example:

```
if (x == 100)
  cout << "x is 100";
else
  cout << "x is not 100";
```

prints on the screen `x is 100` if indeed `x` has a value of `100`, but if it has not -and only if not- it prints out `x is not 100`.

The `if + else` structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the value currently stored in `x` is positive, negative or none of them (i.e. zero):

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

Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces `{ }`.

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

Its format is:

```
while (expression) statement
```

and its functionality is simply to repeat statement while the condition set in expression is true.
For example, we are going to make a program to countdown using a while-loop:

```
// custom countdown using while

#include <iostream>
using namespace std;

int main ()
{
  int n;
  cout << "Enter the starting
number > ";
  cin >> n;

  while (n>0) {
```

```
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

```
    cout << n << ", ";
    --n;
  }

  cout << "FIRE!\n";
  return 0;
}
```

When the program starts the user is prompted to insert a starting number for the countdown. Then the `while` loop begins, if the value entered by the user fulfills the condition `n>0` (that `n` is greater than zero) the block that follows the condition will be executed and repeated while the condition (`n>0`) remains being true.

The whole process of the previous program can be interpreted according to the following script (beginning in main):

1. User assigns a value to `n`
2. The while condition is checked (`n>0`). At this point there are two posibilities:
   * condition is true: statement is executed (to step 3)
   * condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:
   `cout << n << ", ";`
   `--n;`
   (prints the value of `n` on the screen and decreases `n` by `1`)
4. End of block. Return automatically to step 2
5. Continue the program right after the block: print FIRE! and end program.

When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever. In this case we have included `--n;` that decreases the value of the variable that is being evaluated in the condition (`n`) by one - this will eventually make the condition (`n>0`) to become false after a certain number of loop iterations: to be more specific, when `n` becomes `0`, that is where our while-loop and our countdown end.

Of course this is such a simple action for our computer that the whole countdown is performed instantly without any practical delay between numbers.

The do-while loop

Its format is:

```
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, granting at least one execution of `statement` even if `condition` is never fulfilled. For example, the following example program echoes any number you enter until you enter `0`.

```
// number echoer

#include <iostream>
using namespace std;

int main ()
{
```

```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

```
  unsigned long n;
  do {
    cout << "Enter number (0 to
end): ";
    cin >> n;
    cout << "You entered: " << n
<< "\n";
  } while (n != 0);
  return 0;
}
```

The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end. In fact if you never enter the value `0` in the previous example you can be prompted for more numbers forever.

The for loop

Its format is:

```
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. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

It works in the following way:

1. `initialization` is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. `condition` is checked. If it is true the loop continues, otherwise the loop ends and `statement` is skipped (not executed).
3. `statement` is executed. As usual, it can be either a single statement or a block enclosed in braces `{ }`.
4. finally, whatever is specified in the `increase` field is executed and the loop gets back to step 2.

Here is an example of countdown using a for loop:

```
// countdown using a for loop
#include <iostream>
using namespace std;
int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "FIRE!\n";
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
FIRE!
```

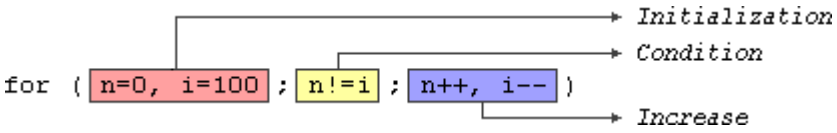
The `initialization` and `increase` fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write:

`for (;n<10;)` if we wanted to specify no initialization and no increase; or `for (;n<10;n++)` if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before).

Optionally, using the comma operator (`,`) we can specify more than one expression in any of the fields included in a `for` loop, like in `initialization`, for example. The comma operator (`,`) is an expression separator, it serves to separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
   // whatever here...
}
```

This loop will execute for 50 times if neither `n` or `i` are modified within the loop:



`n` starts with a value of `0`, and `i` with `100`, the condition is `n!=i` (that `n` is not equal to `i`). Because `n` is increased by one and `i` decreased by one, the loop's condition will become false after the 50th loop, when both `n` and `i` will be equal to `50`.

Q1 Write a program in C++ to print numbers from (1-20) in horizontal line, use (for) command

Q2 Write a program in C++ to print numbers from (1-20) in vertical line, use (for) command

Q3 Write a program in C++ to print odd numbers from

(1-20), use (for) command

Q4 Write a program in C++ to print even numbers from

(1-20), use (for) command

Q5 Write a program in C++ to find (factorial) of any number

Q6 Write a program in C++ to find numbers 7 and 12 from two other numbers mathematically

Q7 Write a program in C++ to sum numbers from (1-10)

Q8 Write a program in C++ to sum the odd numbers from (1-10)

Q9 Write a program in C++ to print numbers from (1-3) five times

Q10 Write a program in C++ to find the smaller numbers between 35 numbers

Q11 Write a program in C++ to find numbers larger than 15 from list of 35 numbers

Q12 Write a program in C++ to print the sum of any numbers except numbers 7

Q13 Write a program in C++ to print Islam 10 times

Q14 Write a program in C++ to solve $v = \sum_{x=1}^{X=20} 1/x$

Q15 Write a program in C++ to solve $\sum_{i=1}^{n} i^2$

Sixth week

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. For example, we are going to stop the count down before its natural end (maybe because of an engine check failure?):

```
// break loop example

#include <iostream>
using namespace std;

int main ()
{
  int n;
  for (n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout       <<       "countdown
aborted!";
      break;
    }
  }
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, countdown
aborted!
```

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. For example, we are going to skip the number 5 in our countdown:

```
// continue loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!
```

```
  }
  cout << "FIRE!\n";
  return 0;
}
```

Functions.

Sin(x)=sinx

Cos(x)=cosx

Tan(x)=tanx

Asin(x)=$\sin^{-1}$

Acos(x)=$\cos^{-1}x$

exp(x)=e^x

sqrt(x)=

pow(x,y)=x^y

Q1write program to find the square root of the numbers from (2-10)

Q2 write program to count down from 10 to 1 and skip number 5 from our count down

Q3 write a program to find the sin of the angles 0 30 60 90 120 150 180 in radian.

Q4 write program to find the root of the second order equation.

Ax2+bx+c=0

Q5 write program in C++ to count down from 10-1 but abort the countdown at number 3

The goto statement

`goto` allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations.
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 (`:`).

Generally speaking, this instruction has no concrete use in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using `goto`:

```
// goto loop 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;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
FIRE!
```

The exit function

`exit` is a function defined in the `cstdlib` library.

The purpose of `exit` is to terminate the current program with a specific exit code. Its prototype is:

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

The `exitcode` is used by some operating systems and may be used by calling programs. By convention, an exit code of `0` means that the program finished normally and any other value means that some error or unexpected results happened.

The selective structure: switch.

The syntax of the switch statement is a bit peculiar. Its objective is to check several possible constant values for an expression. Something similar to what we did at the beginning of this section with the concatenation of several `if` and `else if` instructions. Its form is the following:

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

It works in the following way: 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 (you can include as many `case` labels as values you want to check), the program will execute the statements included after the `default:` label, if it exists (since it is optional).

Both of the following code fragments have the same behavior:

switch example	**if-else equivalent**

`switch (x) {`
`  case 1:`
`  cout << "x is 1";`
`  break;`
`  case 2:`
`  cout << "x is 2";`
`  break;`
`  default:`
`  cout << "value of x unknown";`
`  }`
	`if (x == 1) {`
	`  cout << "x is 1";`
	`  }`
	`else if (x == 2) {`
	`  cout << "x is 2";`
	`  }`
	`else {`
	`  cout << "value of x unknown";`
	`  }`

The `switch` statement is a bit peculiar within the C++ language because it uses labels instead of blocks. This forces us to put `break` statements after the group of statements that we want to be executed for a specific condition. Otherwise the remainder statements -including those corresponding to other labels- will also be executed until the end of the `switch` selective block or a `break` statement is reached.

For example, if we did not include a `break` statement after the first group for case one, the program will not automatically jump to the end of the `switch` selective block and it would continue executing the rest of statements until it reaches either a `break` instruction or the end of the `switch` selective block. This makes unnecessary to include braces `{ }` surrounding the statements for each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression being evaluated. For example:

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x is 1, 2 or 3";
    break;
  default:
    cout << "x is not 1, 2 nor 3";
  }
```

Notice that switch can only be used to compare an expression against constants. Therefore we cannot put variables as labels (for example `case n:` where `n` is a variable) or ranges (`case (1..3):`) because they are not valid C++ constants.

If you need to check ranges or values that are not constants, use a concatenation of `if` and `else if` statements

Q1 Write program in C++ to print the numbers from (1-100) except the numbers can divided on 2,4,6 without remaining

Q2 Write a program to print numbers from (1-10) but if the numbers more than 10 program will print message (the number is more than 10)

Q3 Write program in C++ to print numbers from (1-10) in loop from (1-100)

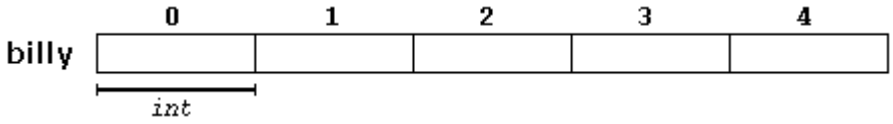
Seventh week

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.

That means that, for example, we can store 5 values of type `int` in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, `int` for example, with a unique identifier.

For example, an array to contain 5 integer values of type `int` called `billy` could be represented like this:



where each blank panel represents an element of the array, that in this case are integer values of type `int`. These elements are numbered from `0` to `4` since in arrays the first index is always `0`, independently of its length.

Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

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

where `type` is a valid type (like `int`, `float`...), `name` is a valid identifier and the `elements` field (which is always enclosed in square brackets `[]`), specifies how many of these elements the array has to contain.

Therefore, in order to declare an array called `billy` as the one shown in the above diagram it is as simple as:

```
int billy [5];
```

NOTE: The `elements` field within brackets `[]` which represents the number of elements the array is going to hold, must be a constant value, since arrays are blocks of non-dynamic memory whose size must be determined before execution. In order to create arrays with a variable length dynamic memory is needed, which is explained later in these tutorials.

Initializing arrays.

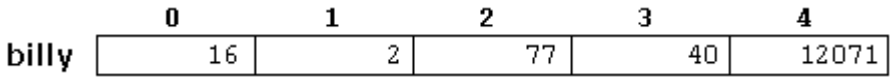
When declaring a regular array of local scope (within a function, for example), if we do not specify otherwise, its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global

and static arrays, on the other hand, are automatically initialized with their default values, which for all fundamental types this means they are filled with zeros.

In both cases, local and global, when we declare an array, we have the possibility to assign initial values to each one of its elements by enclosing the values in braces { }. For example:

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

This declaration would have created an array like this:



The amount of values between braces { } must not be larger than the number of elements that we declare for the array between square brackets [ ]. For example, in the example of array billy we have declared that it has 5 elements and in the list of initial values within braces { } we have specified 5 values, one for each element.

When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty [ ]. In this case, the compiler will assume a size for the array that matches the number of values included between braces { }:

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

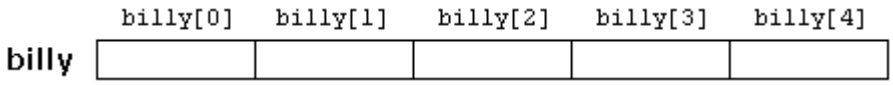
After this declaration, array billy would be 5 ints long, since we have provided 5 initialization values.

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. The format is as simple as:

```
name[index]
```

Following the previous examples in which billy had 5 elements and each of those elements was of type int, the name which we can use to refer to each element is the following:



For example, to store the value 75 in the third element of billy, we could write the following statement:

```
billy[2] = 75;
```

and, for example, to pass the value of the third element of billy to a variable called a, we could write:

```
a = billy[2];
```

Therefore, the expression `billy[2]` is for all purposes like a variable of type `int`.

Notice that the third element of `billy` is specified `billy[2]`, since the first one is `billy[0]`, the second one is `billy[1]`, and therefore, the third one is `billy[2]`. By this same reason, its last element is `billy[4]`. Therefore, if we write billy[5], we would be accessing the sixth element of `billy` and therefore exceeding the size of the array.

In C++ it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause compilation errors but can cause runtime errors. The reason why this is allowed will be seen further ahead when we begin to use pointers.

At this point it is important to be able to clearly distinguish between the two uses that brackets `[ ]` have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements. Do not confuse these two possible uses of brackets `[ ]` with arrays.

```
int billy[5];         // declaration of a new array
billy[2] = 75;        // access to an element of the array.
```

If you read carefully, you will see that a type specifier always precedes a variable or array declaration, while it never precedes an access.

Some other valid operations with arrays:

```
billy[0] = a;
billy[a] = 75;
b = billy [a+2];
billy[billy[a]] = billy[2] + 5;
```

```
// arrays example
#include <iostream>
using namespace std;

int billy [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; n++ )
  {
    result += billy[n];
  }
  cout << result;
  return 0;
}
```

```
12206
```

Q1 find the result of the array 16 2 77 40 12071

Q2 Write a program in C++ to calculate the average of 10 student use array.

Q3 write a program in C++ to print integers number 40 60 50 70 80 by using array then printed inverse order in one line

H.W.

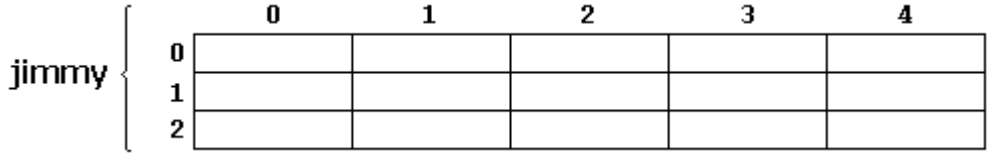
1-Write program to inter 400 students marks then find the deviation of each student mark from the highest one

2-Write program to find the average of 250 students marks and then find the deviation of each mark

Q3 write program in C++ to find the average of 300 students then find how many numbers maximum than average, minimum than average and equal to the average.

Multidimensional arrays

Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a bidimensional table made of elements, all of them of a same uniform data type.

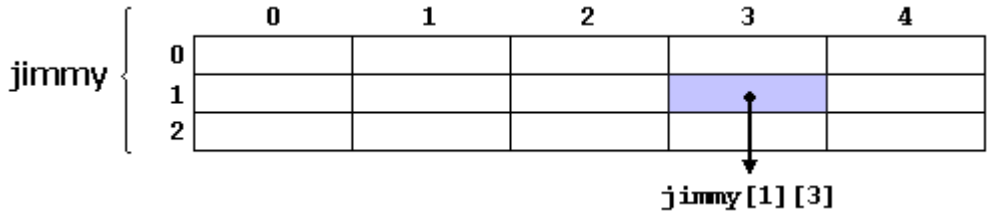


`jimmy` represents a bidimensional array of 3 per 5 elements of type `int`. The way to declare this array in C++ would be:

```
int jimmy [3][5];
```

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

```
jimmy[1][3]
```



(remember that array indices always begin by zero).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. But be careful! The amount of memory needed for an array rapidly increases with each dimension. For example:

```
char century [100][365][24][60][60];
```

declares an array with a `char` element for each second in a century, that is more than 3 billion chars. So this declaration would consume more than 3 gigabytes of memory!

Multidimensional arrays are just an abstraction for programmers, since we can obtain the same results with a simple array just by putting a factor between its indices:

```
int jimmy [3][5];   // is equivalent to
int jimmy [15];     // (3 * 5 = 15)
```

With the only difference that with multidimensional arrays the compiler remembers the depth of each imaginary dimension for us. Take as example these two pieces of code, with both exactly the same result. One uses a bidimensional array and the other one uses a simple array:

multidimensional array	**pseudo-multidimensional array**

`#define WIDTH 5`
`#define HEIGHT 3`
 	
`int jimmy [HEIGHT][WIDTH];`
`int n,m;`
 	
`int main ()`
`{`
`  for (n=0;n<HEIGHT;n++)`
`    for (m=0;m<WIDTH;m++)`
`    {`
`      jimmy[n][m]=(n+1)*(m+1);`
`    }`
`  return 0;`
`}`
	`#define WIDTH 5`
	`#define HEIGHT 3`
	
	`int jimmy [HEIGHT * WIDTH];`
	`int n,m;`
	
	`int main ()`
	`{`
	`  for (n=0;n<HEIGHT;n++)`
	`    for (m=0;m<WIDTH;m++)`
	`    {`
	`      jimmy[n*WIDTH+m]=(n+1)*(m+1);`
	`    }`
	`  return 0;`
	`}`

None of the two source codes above produce any output on the screen, but both assign values to the memory block called jimmy in the following way:

jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

We have used "defined constants" (`#define`) to simplify possible future modifications of the program. For example, in case that we decided to enlarge the array to a height of 4 instead of 3 it could be done simply by changing the line:

```
#define HEIGHT 3
```

to:

```
#define HEIGHT 4
```

with no need to make any other modifications to the program.

Q1 write program in C++ to read the following array and then try to find the sum of the diagonal elements.

```
2  4   6
5  14  2
9  1   8
```

Q2 write program in C++ to sum the upper triangle elements.

```
1  4   6
4  3   10
2  3   6
```

Q3 write a program in C++ to find the sum of the first column in the below matrix.

```
20  5  3
31  17 25
13  4  1
```

H.W Write program in C++ to find the largest number in the matrix below and its location.

```
5   13   22
7   1    50
81  30   9
```

H.W Write program in C++ to split the array below in to (2) one dimension arrays, one contain the odd numbers and the second contain the even numbers.

```
1   2    3   4
5   6    7   8
9  10   11  12
```

Arrays as parameters

At some moment we may need to pass an array to a function as a parameter. 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. In practice this has almost the same effect and it is a much faster and more efficient operation.

In order to accept arrays as parameters the only thing that we have to do when declaring the function is to specify in its parameters the element type of the array, an identifier and a pair of void brackets `[]`. For example, the following function:

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

accepts a parameter of type "array of `int`" called `arg`. In order to pass to this function an array declared as:

```
int myarray [40];
```

it would be enough to write a call like this:

```
procedure (myarray);
```

Here you have a complete example:

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int
length) {
```

```
5 10 15
2 4 6 8 10
```

```
  for (int n=0; n<length; n++)
    cout << arg[n] << " ";
  cout << "\n";
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8,
10};
  printarray (firstarray,3);
  printarray (secondarray,5);
  return 0;
}
```

As you can see, the first parameter (`int arg[]`) accepts any array whose elements are of type `int`, whatever its length. For that reason we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter. This allows the `for` loop that prints out the array to know the range to iterate in the passed array without going out of range.

In a function declaration it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

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

for example, a function with a multidimensional array as argument could be:

```
void procedure (int myarray[][3][4])
```

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.

Arrays, both simple or multidimensional, passed as function parameters are a quite common source of errors for novice programmers. I recommend the reading of the chapter about Pointers for a better understanding on how arrays operate.

Eight week

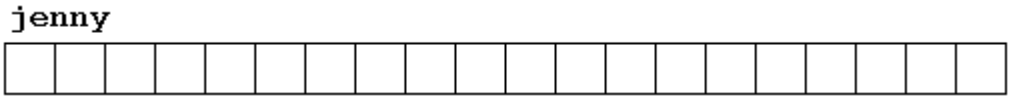
Character Sequences

As you may already know, the C++ Standard Library implements a powerful string class, which is very useful to handle and manipulate strings of characters. However, because strings are in fact sequences of characters, we can represent them also as plain arrays of `char` elements.

For example, the following array:

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

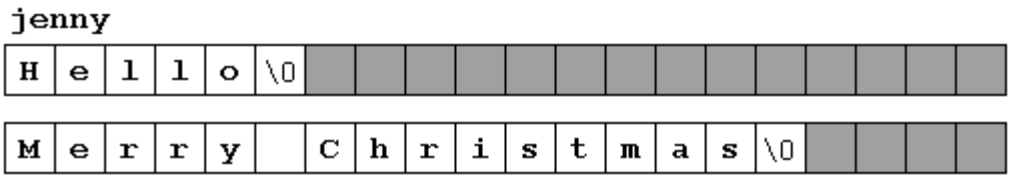
is an array that can store up to 20 elements of type `char`. It can be represented as:



Therefore, in this array, in theory, we can store sequences of characters up to 20 characters long. But we can also store shorter sequences. For example, `jenny` could store at some point in a program either the sequence "`Hello`" or the sequence "`Merry christmas`", since both are shorter than 20 characters.

Therefore, 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'` (backslash, zero).

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



Notice how after the valid content a null character (`'\0'`) has been included in order to indicate the end of the sequence. The panels in gray color represent `char` elements with undetermined values.

Initialization of null-terminated character sequences

Because arrays of characters are ordinary arrays they follow all their same rules. For example, if we want to initialize an array of characters with some predetermined sequence of characters we can do it just like any other array:

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

In this case we would have declared an array of 6 elements of type `char` initialized with the characters that form the word "`Hello`" plus a null character '`\0`' at the end. But arrays of `char` elements have an additional method to initialize their values: using string literals.

In the expressions we have used in some examples in previous chapters, constants that represent entire strings of characters have already showed up several times. These are specified enclosing the text to become a string literal between double quotes ("). For example:

```
"the result is: "
```

is a constant string literal that we have probably used already.

Double quoted strings (") are literal constants whose type is in fact a null-terminated array of characters. So string literals enclosed between double quotes always have a null character ('`\0`') automatically appended at the end.

Therefore 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";
```

In both cases the array of characters `myword` is declared with a size of 6 elements of type `char`: the 5 characters that compose the word "`Hello`" plus a final null character ('`\0`') which specifies the end of the sequence and that, in the second case, when using double quotes (") it is appended automatically.

Please notice that we are talking about initializing an array of characters in the moment it is being declared, and not about assigning values to them once they have already been declared. In fact because this type of null-terminated arrays of characters are regular arrays we have the same restrictions that we have with any other array, so we are not able to copy blocks of data with an assignment operation.

Assuming `mystext` is a `char[]` variable, expressions within a source code like:

```
mystext = "Hello";
mystext[] = "Hello";
```

would not be valid, like neither would be:

```
mystext = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

The reason for this may become more comprehensible once you know a bit more about pointers, since then it will be clarified that an array is in fact a constant pointer pointing to a block of memory.

Using null-terminated sequences of characters

Null-terminated sequences of characters are the natural way of treating strings in C++, so they can be used as such in many procedures. In fact, regular string literals have this type (`char[]`) and can also be used in most cases.

For example, `cin` and `cout` support null-terminated sequences as valid containers for sequences of characters, so they can be used directly to extract strings of characters from `cin` or to insert them into `cout`. For example:

```
// null-terminated sequences of characters
#include <iostream>
using namespace std;

int main ()
{
  char question[] = "Please, enter your first name: ";
  char greeting[] = "Hello, ";
  char yourname [80];
  cout << question;
  cin >> yourname;
  cout << greeting << yourname << "!";
  return 0;
}
```

```
Please, enter your first name: John
Hello, John!
```

As you can see, we have declared three arrays of `char` elements. The first two were initialized with string literal constants, while the third one was left uninitialized. In any case, we have to speficify the size of the array: in the first two (`question` and `greeting`) the size was implicitly defined by the length of the literal constant they were initialized to. While for `yourname` we have explicitly specified that it has a size of 80 chars.

Finally, 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;
```

NINE WEEK

Graphics

getmaxx and getmaxy <GRAPHICS.H>

Declaration:

- int far getmaxx(void);
- int far getmaxy(void);

Remarks:

- getmaxx returns the maximum x value (screen-relative) for the current graphics driver and mode.
- getmaxy returns the maximum y value (screen-relative) for the current graphics driver and mode.

getx, gety <GRAPHICS.H>

Declaration:

- int far getx(void);
- int far gety(void);

Remarks:

- getx returns the x-coordinate of the current graphics position.
- gety returns the y-coordinate of the current graphics position.

moverel, moveto <GRAPHICS.H>

Declaration:

- void far moverel(int dx, int dy);
- void far moveto(int x, int y);

Remarks:

- moverel moves the current position (CP) dx pixels in the x direction and dy pixels in the y direction.

- moveto moves the current position (CP) to viewport position (x, y).

getpixel, putpixel <GRAPHICS.H>

Declaration:

- unsigned far getpixel(int x, int y);
- void far putpixel(int x, int y, int color);

Remarks:

- getpixel gets the color of the pixel located at (x,y).
- putpixel plots a point in the color defined by color at (x,y).

line, linerel, lineto <GRAPHICS.H>

Declaration:

- void far line(int x1, int y1, int x2, int y2);
- void far linerel(int dx, int dy);
- void far lineto(int x, int y);

Remarks:

- line draws a line from (x1, y1) to (x2, y2) using the current color, line style, and thickness. It does not update the current position (CP).

- linerel draws a line from the CP to a point that is a relative distance (dx, dy) from the CP, then advances the CP by (dx, dy).

- lineto draws a line from the CP to (x, y), then moves the CP to (x, y).

arc, circle, pieslice <GRAPHICS.H>

Declaration:

- void far arc(int x, int y, int stangle, int endangle, int radius);
- void far circle(int x, int y, int radius);
- void far pieslice(int x, int y, int stangle, int endangle, int radius);

Remarks:

arc draws a circular arc in the current drawing color.

circle draws a circle in the current drawing color.

pieslice draws a pie slice in the current drawing color, then fills it using the current fill pattern and fill color.

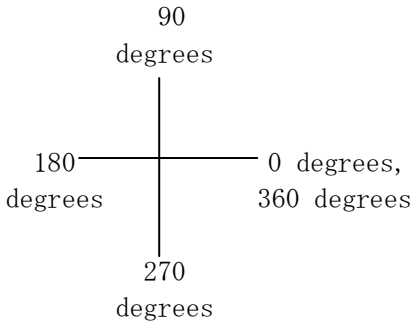
Argument	What It Is/Does
(x,y)	Center point of arc, circlew, or pie slice
stangle	Start angle in degrees
endangle	End angle in degrees
radius	Radius of arc, circle, and pieslice

The arc or slice travels from stangle to endangle.

If stangle = 0 and endangle = 360, the call to arc draws a complete circle.

If your circles are not perfectly round, use set aspect ratio to adjust the aspect ratio.

Angle for arc, circle, and pieslice (counter-clockwise)



The linestyle parameter does not affect arcs, circles, ellipses, or pie slices. Only the thickness parameter is used.

rectangle <GRAPHICS.H>

Declaration:

void far rectangle(int left, int top, int right, int bottom);

Remarks:

rectangle draws a rectangle in the current line style, thickness, and drawing color.

(left,top) is the upper left corner of the rectangle, and (right,bottom) is its lower right corner.

ellipse, fillellipse, sector <GRAPHICS.H>

Declaration:

- void far ellipse(int x, int y, int stangle, int endangle, int xradius, int yradius);
- void far fillellipse(int x, int y,int xradius, int yradius);
- void far sector(int x, int y, int stangle, int endangle,int xradius, int yradius);

Remarks:

ellipse draws an elliptical arc in the current drawing color.

fillellipse draws an ellipse, then fills the ellipse with the current fill color and fill pattern.

sector draws and fills an elliptical pie slice in the current drawing color, then fills it using the pattern and color defined by setfillstyle or setfillpattern.

Argument : What It Is

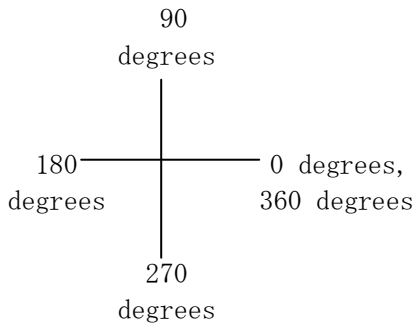
(x,y) : Center of ellipse
xradius : Horizontal axis
yradius : Vertical axis
stangle : Starting angle

endangle : Ending angle

The ellipse or sector travels from stangle to endangle.

If stangle = 0 and endangle = 360, the call to ellipse draws a complete ellipse.

Angle for ellipse, fillellipse, and sector (counter-clockwise)



The linestyle parameter does not affect arcs, circles, ellipses, or pie slices. Only the thickness parameter is used.

bar <GRAPHICS.H>

Declaration: void far bar(int left, int top, int right, int bottom);
Remarks:
bar draws a filled-in, rectangular, two-dimensional bar.
The bar is filled using the current fill pattern and fill color. bar does not outline the bar.
To draw an outlined two-dimensional bar, use bar3d with depth = 0.

Parameters : What they are
(left, top) : the rectangle's upper left corner
(right, bottom) : the rectangle's lower right corner
The coordinates are in pixels.

getcolor, setcolor <GRAPHICS.H>

Declaration:
- int far getcolor(void);
- void far setcolor(int color);

Remarks:
getcolor returns the current drawing color.
setcolor sets the current drawing color to color, which can range from 0 to getmaxcolor.
To select a drawing color with setcolor, you can pass either the color number or the equivalent color name.
The drawing color is the value that pixels are set to when the program draws lines, etc.

COLORS (text mode)

Constant	Value	Back-grnd?	Fore-grnd

BLACK	: 0	: Yes	: Yes
BLUE	: 1	: Yes	: Yes
GREEN	: 2	: Yes	: Yes
CYAN	: 3	: Yes	: Yes
RED	: 4	: Yes	: Yes
MAGENTA	: 5	: Yes	: Yes
BROWN	: 6	: Yes	: Yes
LIGHTGRAY	: 7	: Yes	: Yes
DARKGRAY	: 8	: No	: Yes
LIGHTBLUE	: 9	: No	: Yes
LIGHTGREEN	: 10	: No	: Yes
LIGHTCYAN	: 11	: No	:Yes
LIGHTRED	: 12	: No	:Yes
LIGHTMAGENTA	: 13	: No	: Yes
YELLOW	: 14	: No	: Yes
WHITE	: 15	: No	: Yes
BLINK	:128	: No	: ***

*** To display blinking characters in text mode, add BLINK to the foreground color. (Defined in CONIO.H.)

getmaxcolor <GRAPHICS.H>

Returns maximum color value

Declaration: int far getmaxcolor(void);

Remarks:
getmaxcolor returns the highest valid color value that can be passed to setcolor for the current graphics driver and mode.

getbkcolor, setbkcolor <GRAPHICS.H>

Declaration:

```
int far getbkcolor(void);
void far setbkcolor(int color);
```

Remarks:
getbkcolor returns the current background color.

setbkcolor sets the background to the color specified by color.

cleardevice <GRAPHICS.H>

Clears the graphics screen

Declaration: void far cleardevice(void);

Remarks:
cleardevice erases the entire graphics screen and moves the CP (current
position) to home (0,0).

(Erasing consists of filling with the current background color.)

setlinestyle <GRAPHICS.H>

Sets the current line style and width or pattern
Declaration:
  void far setlinestyle(int linestyle, unsigned upattern, int thickness);

Remarks:
setlinestyle sets the style for all lines drawn by line, lineto, rectangle, drawpoly, etc.

line_styles <GRAPHICS.H>

Enum: Line styles for getlinesettings and setlinestyle.

Name	Value	Meaning
SOLID_LINE	0	Solid line
DOTTED_LINE	1	Dotted line
CENTER_LINE	2	Centered line
DASHED_LINE	3	Dashed line
USERBIT_LINE	4	User-defined line style

line_widths <GRAPHICS.H>

Enum: Line widths for getlinesettings and setlinestyle.

Name	Value	Meaning
NORM_WIDTH	1	1 pixel wide
THICK_WIDTH	3	3 pixels wide

setfillstyle <GRAPHICS.H>

Declaration: void far setfillstyle(int pattern, int color);

Remarks:
setfillstyle sets the current fill pattern and fill color.

To set a user-defined fill pattern, do not give a pattern of 12 (USER_FILL)to setfillstyle; instead, call setfillpattern.

The enumeration fill_patterns, defined in GRAPHICS.H, gives names for the predefined fill patterns, plus an indicator for a user-defined pattern.

fill_patterns <GRAPHICS.H>

Enum: Fill patterns for getfillsettings and setfillstyle.

Names	Value	Means Fill With...
EMPTY_FILL	0	Background color
SOLID_FILL	1	Solid fill
LINE_FILL	2	---
LTSLASH_FILL	3	///

```
SLASH_FILL      :  4  : ///, thick lines
BKSLASH_FILL    :  5  : \\\, thick lines
LTBKSLASH_FILL  :  6  : \\\
HATCH_FILL      :  7  : Light hatch
XHATCH_FILL     :  8  : Heavy crosshatch
INTERLEAVE_FILL :  9  : Interleaving lines
WIDE_DOT_FILL   : 10  : Widely spaced dots
CLOSE_DOT_FILL  : 11  : Closely spaced dots
USER_FILL       : 12  : User-defined fill pattern
```

All but EMPTY_FILL fill with the current fill color. EMPTY_FILL uses the current background color.

floodfill <GRAPHICS.H>

Declaration: void far floodfill(int x, int y, int border);

Remarks:

floodfill fills an enclosed area on bitmap devices.

The area bounded by the color border is flooded with the current fill pattern and fill color.

(x,y) is a "seed point".

- If the seed is within an enclosed area, the inside will be filled.
- If the seed is outside the enclosed area, the exterior will be filled.

Use fillpoly instead of floodfill whenever possible so you can maintain code compatibility with future versions.

Example:

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main(void)
{
   /* request auto detection */
   int gdriver = DETECT, gmode, errorcode;
   int maxx, maxy;

   /* initialize graphics, local variables
*/
   initgraph(&gdriver, &gmode, "");

   /* read result of initialization */
   errorcode = graphresult();
   if (errorcode != grOk)
   /* an error occurred */
   {
      printf("Graphics error: %s\n", grapherrormsg(errorcode));
      printf("Press any key to halt:");
      getch();
      exit(1);
      /* terminate with an error code */
   }

   maxx = getmaxx();
   maxy = getmaxy();
```

```
    /* select drawing color */
    setcolor(getmaxcolor());

    /* select fill color */
    setfillstyle(SOLID_FILL, getmaxcolor());

    /* draw a border around the screen */
    rectangle(0, 0, maxx, maxy);

    /* draw some circles */
    circle(maxx / 3, maxy /2, 50);
    circle(maxx / 2, 20, 100);
    circle(maxx-20, maxy-50, 75);
    circle(20, maxy-20, 25);

    /* wait for a key */
    getch();

    /* fill in bounded region */
    floodfill(2, 2, getmaxcolor());

    /* clean up */
    getch();
    closegraph();
    return 0;
}
```

textheight, textwidth <GRAPHICS.H>

Declaration:

- int far textheight(char far *textstring);
- int far textwidth(char far *textstring);

Remarks:

- textheight takes the current font size and multiplication factor, and determines the height of textstring in pixels.

- textwidth takes the string length, current font size, and multiplication factor, and determines the width of textstring in pixels.

settextstyle <GRAPHICS.H>

Declaration:

void far settextstyle(int font, int direction, int charsize);

Remarks:

settextstyle sets the text font, the direction in which text is displayed, and the size of the characters.

A call to settextstyle affects all text output by outtext and outtextxy.

font

One 8x8 bit-mapped font and several "stroked" fonts are available. The 8x8 bit-mapped font, the default, is built into the graphics system.

The enumeration font_names, defined in GRAPHICS.H, provides names for the different font settings.

font_names <GRAPHICS.H>

Enum: Names for BGI fonts

Name	Value	Meaning
DEFAULT_FONT	0	8x8 bit-mapped font
TRIPLEX_FONT	1	Stroked triplex font
SMALL_FONT	2	Stroked small font
SANS_SERIF_FONT	3	Stroked sans-serif font
GOTHIC_FONT	4	Stroked gothic font

direction

Font directions supported are horizontal text (left to right) and vertical text (rotated 90 degrees counterclockwise).

The default direction is HORIZ_DIR.

Name	Value	Direction
HORIZ_DIR	0	Left to right
VERT_DIR	1	Bottom to top

charsize

The size of each character can be magnified using the charsize factor. If charsize is non-zero, it can affect bit-mapped or stroked characters.

outtext, outtextxy <GRAPHICS.H>

Declaration:

- void far outtext(char far *textstring);
- void far outtextxy(int x, int y, char far *textstring);

Remarks:

outtext and outtextxy display a text string, using the current justification settings and the current font, direction, and size.

- outtext outputs textstring at the current position (CP)
- outtextxy displays textstring in the viewport at the position (x, y)

To maintain code compatibility when using several fonts, use textwidth and textheight to determine the dimensions of the string.
If a string is printed with the default font using outtext or outtextxy, any part of the string that extends outside the current viewport is truncated.
With outtext, if the horizontal text justification is LEFT_TEXT and the text direction is HORIZ_DIR, the CP's x-coordinate is advanced by textwidth(textstring).
Otherwise, the CP remains unchanged.
outtext and outtextxy are for use in graphics mode; they will not work in text mode.

outtext example

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main(void)
```

```
{
   /* request auto detection */
   int gdriver = DETECT, gmode, errorcode;
   int midx, midy;

   /* initialize graphics and local variables */
   initgraph(&gdriver, &gmode, "");

   /* read result of initialization */
   errorcode = graphresult();
   if (errorcode != grOk)  /* an error occurred */
   {
      printf("Graphics error: %s\n", grapherrormsg(errorcode));
      printf("Press any key to halt:");
      getch();
      exit(1); /* terminate with an error code */
   }

   midx = getmaxx() / 2;
   midy = getmaxy() / 2;
   /* move the C.P. to the center of the screen */
   moveto(midx, midy);

   /* output text starting at the C.P. */
   outtext("This ");
   outtext("is ");
   outtext("a ");
   outtext("test.");
   /* clean up */
   getch();
   closegraph();
   return 0;
}
```

getimage, putimage <GRAPHICS.H>

- getimage saves a bit image of the specified region into memory
- putimage outputs a bit image onto the screen

Declaration:

- void far getimage(int left, int top, int right, int bottom,void far *bitmap);
- void far putimage(int left, int top, void far *bitmap, int op);

Remarks:

- getimage copies an image from the screen to memory.

- putimage puts the bit image previously saved with getimage back onto the screen, with the upper left corner of the image placed at (left,top).

Argument	What It Is/Does
bitmap	Points to the area in memory where the bit image is stored. The first two words of this area are used for the width and height of the rectangle. The remainder holds the Image itself.
bottom left right top	(left, top) and (right, bottom) define the rectangular screen area from which getimage copies the bit image. (left, top) is where putimage places the upper left corner of the stored image.
op	Specifies a combination operator that controls how the color for each destination pixel onscreen is computed, based on the pixel already onscreen and the corresponding source pixel in memory.

The enumeration putimage_ops, defined in GRAPHICS.H, gives names to the putimage combination operators.

putimage_ops <GRAPHICS.H>

Enum: Operators for putimage

Constant:	Value	: Meaning
COPY_PUT :	0	: Copies source bitmap onto screen
XOR_PUT :	1	: Exclusive ORs source image with that already onscreen
OR_PUT :	2	: Inclusive ORs image with that already onscreen
AND_PUT :	3	: ANDs image with that already onscreen
NOT_PUT :	4	:Copy the inverse of the source

imagesize <GRAPHICS.H>

Declaration:

unsigned far imagesize(int left, int top, int right, int bottom);

Remarks:

imagesize determines the size of the memory area required to store a bit image.

Return Value:

- On success, returns the size of the required memory area in bytes.
- On error (if the size required for the selected image is >= (64K - 1) bytes),
  returns 0xFFFF (-1)

getimage example

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <alloc.h>

void save_screen(void far *buf[4]);
void restore_screen(void far *buf[4]);

int maxx, maxy;

int main(void)
{
   int gdriver=DETECT, gmode, errorcode;
   void far *ptr[4];

   /* auto-detect the graphics driver and mode */
   initgraph(&gdriver, &gmode, "");
   errorcode = graphresult(); /* check for any errors */
   if (errorcode != grOk)
   {
      printf("Graphics error: %s\n", grapherrormsg(errorcode));
      printf("Press any key to halt:");
      getch();
      exit(1);
   }
   maxx = 50; //getmaxx();
   maxy = 60;  //getmaxy();

   /* draw an image on the screen */
   rectangle(0, 0, maxx, maxy);
   line(0, 0, maxx, maxy);
   line(0, maxy, maxx, 0);
   save_screen(ptr);    /* save the current screen */
   getch();             /* pause screen */
   cleardevice();       /* clear screen */
   getch();
   restore_screen(ptr); /* restore the screen */
   getch();             /* pause screen */
```

```
   closegraph();
   return 0;
}
void save_screen(void far *buf[4])
{
   unsigned size;
   int ystart=0, yend, yincr, block;
   yincr = (maxy+1) / 4;
   yend = yincr;
   size = imagesize(0, ystart, maxx, yend);
/* get byte size of image */

   for (block=0; block<=3; block++)
   {
      if ((buf[block] = farmalloc(size)) == NULL)
      {
         closegraph();
         printf("Error: not enough heap space in save_screen().\n");
         exit(1);
      }
      getimage(0, ystart, maxx, yend, buf[block]);
      ystart = yend + 1;       yend += yincr + 1;
   }
}
void restore_screen(void far *buf[4])
{
   int ystart=0, yend, yincr, block;
   yincr = (maxy+1) / 4;
   yend = yincr;

   for (block=0; block<=3; block++)
   {
      putimage(0, ystart, buf[block], COPY_PUT);
      farfree(buf[block]);
      ystart = yend + 1;       yend += yincr + 1;
   }
}
```

Applications on Graphics

Q) Draw football stadd

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main(void)
{
   /* request auto detection */
   int gdriver = DETECT, gmode, errorcode;

   /* initialize graphics mode */
   initgraph(&gdriver, &gmode, "");

   /* read result of initialization */
   errorcode = graphresult();

   if (errorcode != grOk)  /* an error occurred */
   {
      printf("Graphics error: %s\n", grapherrormsg(errorcode));
      printf("Press any key to halt:");       getch();
      exit(1);             /* return with error code */
   }

   /* draw green bar with white border */
```

```
/*   setfillstyle(1, 2);
   bar(100, 100, 500, 300);
   setcolor(15);
   rectangle(100, 100, 500, 300);*/
 //or
   setcolor(15);
   rectangle(100, 100, 500, 300);
   setfillstyle(1, 2);
   floodfill(200,200,15);

   line(300,100,300,300);
   circle(300,200,30);
   rectangle(100,150,160,250);
   rectangle(100,170,130,230);
   rectangle(500,150,440,250);
   rectangle(500,170,470,230);
   arc(160,200,270,90,20);
   arc(440,200,90,270,20);

   arc(100,100,270,360,10);
   arc(500,100,180,270,10);
   arc(100,300,0,90,10);
   arc(500,300,90,180,10);

   /* clean up */
   getch();
   closegraph();
   return 0;
}
```

Q) Draw Tanis stadd

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main(void)
{
   /* request auto detection */
   int gdriver = DETECT, gmode, errorcode;

   /* initialize graphics mode */
   initgraph(&gdriver, &gmode, "");

   /* read result of initialization */
   errorcode = graphresult();

   if (errorcode != grOk)  /* an error occurred */
   {
      printf("Graphics error: %s\n", grapherrormsg(errorcode));
      printf("Press any key to halt:");
      getch();
      exit(1);             /* return with error code */
   }

   /* draw green bar with white border */
/*   setfillstyle(1, 2);
```

```
    bar(100, 100, 400, 300);
    setcolor(15);
    rectangle(100, 100, 400, 300);*/
  //or
    setcolor(15);
    rectangle(100, 100, 400, 300);
    setfillstyle(1, 2);
    floodfill(200,200,15);

    rectangle(100, 150, 400, 250);
    rectangle(175, 150, 325, 250);

    setfillstyle(1, 15);
    circle(500,200,10);
    floodfill(500,200,15);
    bar(452,180,458,250);

    setfillstyle(8, 14);
    ellipse(455,160,0,360,10,20);
    floodfill(455,160,15);


    /* clean up */
    getch();
    closegraph();
    return 0;
}
```