**Dr. Abd El-Aziz Ahmed**

**The Features of C++ as a Language**

1. **High level language**: A high-level language focuses more on concepts that are easy to understand by the human mind, such as objects or mathematical functions. A *low-level language* is generally quite similar to machine code, and thus is more suitable for programs like device drivers or very high performance programs that really need access to the hardware.
2. **Compiled language**: C++ is compiled directly to a machine's native code, allowing it to be one of the fastest languages in the world, if optimized. *Compiled languages* are translated to the target machine's native language by a program called a compiler. This can result in very fast code, especially if the compiler is effective at optimizing, however the resulting code may not port well across operating systems and the compilation process may take a while. *Interpreted language****s*** are read by a program called an interpreter line by line and are executed by that program. While they are as portable as their interpreter and have no long compile times, interpreted languages are usually *much* slower than an equivalent compiled program.  Finally, *just-in-time compiled* (or JIT-compiled) languages are languages that are quickly compiled when programs written in them need to be run (usually with very little optimization), offering a balance between performance and portability.
3. **Portable:** As one of the most frequently used languages in the world and as an open language, C++ has a wide range of compilers that run on many different platforms that support it. Code that exclusively uses C++'s standard library will run on many platforms with few to no changes.
4. **Sensitive:**  It differentiates between the lowercase and uppercase. All the keywords are lowercase.

**Definitions**

**Compiler**: it reads, analyses, and translates the source code into an object code.



**Example:**

Firstprogram.cpp and secondprogram.cpp are two programs compiled; hence, a new file (object file) will be created that ends with .obj, such as: firstprogram.obj and secondprogram.obj The compiler also catches grammatical errors called syntax errors in the source

**Linker:** after the compile process is completed, the computer must do one more thing before we have a copy of the machine code that is ready to be executed. Most programs are not entirely complete in and of themselves. They need other modules previously written that perform certain operations such as data input and output. Our programs need these attachments in order to run. This is the function of the linking process it. The linker combines one or more objects files and library code into executable, library, or list of error messages. The linker goes to a “software library” of programs and attaches the appropriate code to your program. This produces what is called the executable code, generated in a file that often ends with .exe.



**Loader:** it reads the exe file into the memory.

**Structure of a C++ Program.**

// my first program in C++

#include <iostream.h>

int main()

{

 cout << "Hello World!";

 return 0;

}

Line 1: // my first program in C++

Two slash signs indicate that the rest of the line is a comment inserted by the programmer but which has no effect on the behavior of the program. Programmers use them to include short explanations or observations concerning the code or program. In this case, it is a brief introductory description of the program.

Line 2: #include <iostream.h>

Lines beginning with a hash sign (#) are directives read and interpreted by what is known as the *preprocessor*. They are special lines interpreted before the compilation of the program itself begins. In this case, the directive #include <iostream.h>, instructs the preprocessor to include a section of standard C++ code, known as *header iostream*.h, that allows to perform standard input and output operations, such as writing the output of this program (Hello World) to the screen.

Line 3: A blank line.

Blank lines have no effect on a program. They simply improve readability of the code.

Line 4: int main ()

This line initiates the declaration of a function. Essentially, a function is a group of code statements which are given a name: in this case, this gives the name "main" to the group of code statements that follow. Functions will be discussed in detail in a later chapter, but essentially, their definition is introduced with a succession of a type (int), a name (main) and a pair of parentheses (()), optionally including parameters. The function named main is a special function in all C++ programs; it is the function called when the program is run. The execution of all C++ programs begins with the main function, regardless of where the function is actually located within the code.

Lines 5 and 7: { and }

The open brace ({) at line 5 indicates the beginning of main's function definition, and the closing brace (}) at line 7, indicates its end. Everything between these braces is the function's body that defines what happens when main is called. All functions use braces to indicate the beginning and end of their definitions.

Here you have a list of the single character escape codes:

|  |  |
| --- | --- |
| **Escape code** | **Description** |
| \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 (\) |

**Cin>> command**

Cin>>x;

**Variables and Types**

The *variable is* a portion of memory to store a value. Each variable needs a name that identifies it and distinguishes it from the others and a *type*.

**Declaration:**name type;

*Name*: When naming items in C++, you need to observe the following rules:

* The first character of a name must be a letter or an underscore (\_).
* Subsequent characters may be underscores, letters, or digits.
* Identifiers in C++ are case-sensitive. For example, the names volume, VOLUME, VOLume, and Volume are four different identifiers.
* You cannot use reserved words, such as int, double, or static, as identifiers.

Here are examples of valid identifiers:

y

x

myString

HOURS\_PER\_DAY

HexNumber1

hex\_number\_1

hex1Number3

\_Length

\_length\_

*Fundamental data types* are basic types implemented directly by the language that represent the basic storage units supported natively by most systems. They can mainly be classified into:

* Character types: They can represent a single character, such as 'A' or '$'. The most basic type is char, which is a one-byte character. Other types are also provided for wider characters.
* Numerical integer types: They can store a whole number value, such as 7 or 1024. They exist in a variety of sizes, and can either be signed or unsigned, depending on whether they support negative values or not.
* Floating-point types: They can represent real values, such as 3.14 or 0.01, with different levels of precision, depending on which of the three floating-point types is used.
* Boolean type: The boolean type, known in C++ as bool, can only represent one of two states, true or false.

Here is the complete list of fundamental types in C++:

|  |  |  |
| --- | --- | --- |
| **Group** | **Type names\*** | **Notes on size / precision** |
| Character types | **Char** | Exactly one byte in size. At least 8 bits. |
| **char16\_t** | Not smaller than char. At least 16 bits. |
| **char32\_t** | Not smaller than char16\_t. At least 32 bits. |
| **wchar\_t** | Can represent the largest supported character set. |
| Integer types (signed) | **signed char** | Same size as char. At least 8 bits. |
| *signed* **short** *int* | Not smaller than char. At least 16 bits. |
| *signed* **int** | Not smaller than short. At least 16 bits. |
| *signed* **long** *int* | Not smaller than int. At least 32 bits. |
| *signed* **long long** *int* | Not smaller than long. At least 64 bits. |
| Integer types (unsigned) | **unsigned char** | (same size as their signed counterparts) |
| **unsigned short** *int* |
| **unsigned** *int* |
| **unsigned long** *int* |
| **unsigned long long** *int* |
| Floating-point types | **Float** |  |
| **Double** | Precision not less than float |
| **long double** | Precision not less than double |
| Boolean type | **Bool** |  |
| Void type | **Void** | no storage |
| Null pointer | **decltype(nullptr)** |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Data Type | Byte Size | Range | Examples |
| Bool | 1 | false and true | false, true |
| Char | 1 | –128 to 127 | 'A','@' |
| signed char | 1 | –128 to 127 | 23 |
| unsigned char | 1 | 0 to 255 | 250,0x1c |
| int (16-bit) | 2 | –32768 to 32767 | 3200, –6000 |
| int (32-bit) | 4 | -2147483648 to 2147483647 | –1000000, 345678 |
| unsigned int (16 bit) | 2 | 0 to 56635 | 0x00aa, 32769 |
| unsigned int (32-bit) | 4 | 0 to 4294967295 | 0xffea, 65535 |
| short int | 2 | –32768 to 32767 | 234 |
| unsigned short int | 2 | 0 to 65535 | 0x1e, 52000 |
| long int | 4 | –2147483648 to 2147483647 | 0xaffaf, –64323 |
| unsigned long int | 4 | 0 to 4294967295 | 167556 |
| Float | 4 | 3.4E–38 to 3.4E+38 and –3.4E–38 to –3.4E+38 | –15.443, 22.35, 2.45e+24 |
| Double | 8 | 1.7E–308 to 1.7E+308 and –1.7E–308 to –1.7E+308 | –2.5e+100,   –78.32544 |
| long double | 10 | 3.4E–4932 to 1.1E+4932 and –1.1E–4932 to –3.4E+4932 | 8.5e–3000, –9.345e+2341 |

**Definition**: The definition of the variable is done, when it has a value and knows where it will be stored.

**Initialization:** when a variable has a specific value from the moment it is declared. This is called the initialization of the variable. There are three ways for the initialization:

type identifier = initial\_value;

type identifier (initial\_value);

type identifier {initial\_value};

**Constants**

C++ allows you to declare constants either using the #**define** directive or using the formal constant syntax. The general syntax for declaring a formal constant is:

const *type* *constantName* = *constantValue*;

The declaration of a constant resembles the declaration of an initialized variable. Declaring a constant requires the keyword **const**. If you omit the constant's type, the compiler uses the **int** data type.

Here are examples of constants:

const int MAX\_NUM = 1000;

const int MIN\_NUM = 1;

const SEC\_PER\_MINUTE = 60;

const char FIRST\_DRIVE = 'A';

const double MIN\_RATE = 0.023;

**Using #define**

#define constantName constant value

For Example: #define ARRAY\_SIZE 100

**Arithmetic operators**

|  |  |  |  |
| --- | --- | --- | --- |
| *C++ Operator* | *Role* | *Data Type* | *Example* |
| + | unary plus | numerical | z = +h – 2 |
| - | unary minus | numerical | z = –1 \* (z+1) |
| + | Add | numerical | h = 34 + g |
| - | subtract | numerical | z = 3.4 – t |
| / | divide | numerical | d = m / v |
| \* | multiply | numerical | area = len \* wd |
| % | modulus | integers | count = w % 12 |

Example:

#include <iostream.h>

main()

{

  int nNum1, nNum2;

  long lAdd, lSub, lMul, lDiv, lMod;

  double fX, fY;

  double fAdd, fSub, fMul, fDiv;

  // prompt for two integers

  cout << "Enter a nonzero integer : ";

  cin >> nNum1;

  cout << "Enter another nonzero integer : ";

  cin >> nNum2;

  cout << "\n";

  // apply arithmetic operators

  lAdd = nNum1 + nNum2;

  lSub = nNum1 - nNum2;

  lMul = nNum1 \* nNum2;

  lDiv = nNum1 / nNum2;

  lMod = nNum1 % nNum2;

  // display operands and results

  cout << nNum1 << " + " << nNum2 << " = " << lAdd << "\n";

  cout << nNum1 << " - " << nNum2 << " = " << lSub << "\n";

  cout << nNum1 << " \* " << nNum2 << " = " << lMul << "\n";

  cout << nNum1 << " / " << nNum2 << " = " << lDiv << "\n";

  cout << nNum1 << " % " << nNum2 << " = " << lMod << "\n";

  cout << "\n";

return 0;

}

**Increment / Decrement Operators**

C++ offers the increment operators **++** and **--** to support a shorthand syntax for adding or subtracting 1 from the value in a variable, respectively. The general syntax for the operator **++** is:

// form 1: preincrement (prefix)

 ++variableName

// form 2: postincrement (postfix)

 variableName++

As for the decrement operator, the general syntax for this operator is:

// form 1: pre-decrement

 --variableName

// form 2: post-decrement

 variableName--

Suppose j = 10;

|  |  |  |
| --- | --- | --- |
| *Expression* | *Evaluated As* | *Result* |
| j \* j \* j++ | 10 \* 10 \* 10 | 1000 |
| j \* j++ \* j | 10 \* 10 \* 10 | 1000 |
| j++ \* j \* j | 10 \* 10 \* 10 | 1000 |
| j \* j \* ++j | 10 \* 10 \* 11 | 1100 |
| j \* ++j \* j | 11 \* 11 \* 11 | 1331 |
| ++j \* j \* j | 11 \* 11 \* 11 | 1331 |
| --j\*10+++j | j=9, then90+10 | 100 |
| --j\*(10+++j) | j=9, then10\*(10+10) | 200 |

If we use a constant, the prefix is executed before the multiplication, operation, because the compiler executes the prefix before the multiplication if there’s a constant operand of the multiplication operation.

Example:

Let k=2, n=10,

2\*n+ ++n= 2\*11+11=33

k\*n+ ++n= 2\*10+11=31

**Assignment Operators**

|  |  |  |
| --- | --- | --- |
| *C++ Operator* | *Example* | *Long-Form Example* |
| += | fSum += fX; | fSum = fSum + fX; |
| –= | fY –= fX; | fY = fY – fX; |
| /= | nCount /= N; | nCount = nCount / N; |
| \*= | fScl \*= fFcator; | fScl = fScl \* fFactor; |
| %= | nBins %= nCount; | nBins = nBins % nCount; |

**Relational Operators**

|  |  |  |
| --- | --- | --- |
| ***C++ Operator*** | ***Meaning*** | ***Example*** |
| < | Less than | k < 12 |
| <= | Less than or equal to | k <= 33 |
| > | greater than | k > 45 |
| >= | greater than or equal to | k >= 77 |
| == | equal to | k == 32 |
| != | not equal to | k != 33 |
| ?: | conditional assignment | k = (k < 0)? 1 : k |

**Logical Operators**

|  |  |  |
| --- | --- | --- |
| ***C++ Operator*** | ***Meaning*** | ***Example*** |
| && | logical AND | k > 1 && k < 11 |
| || | logical OR | k < 0 || k > 22 |
| ! | logical NOT | !(k > 1 && k < 10) |

**Note:** every number (positive or negative) is true except zero (0) is false.

**Example:**

41 < 65 is TRUE

41 <= 29 is FALSE

65 > 29 is TRUE

41 == 65 is FALSE

41 != 29 is TRUE

41 < 65 && 65 < 29 is FALSE

41 < 65 || 65 < 29 is TRUE

!(41 <= 65 ) is FALSE

(0) is FALSE

(45) is TRUE

!(14%7) && !(10%7) is FALSE

!(-9) is FALSE

**--**x||y &&z is ( (--x)||(y&&z)).

**++**x+y>!z/w is ( ((++x)+y) > ((!z)/w) ).

**Precedence of arithmetic operators.**

|  |  |  |
| --- | --- | --- |
| **Operator(s)** | **Operation(s)** | **Order of evaluation (precedence)** |
| ++/-- Prefix  |  | Right-to-left. The operator is executed first, when it’s involved in an arithmetic operation. |
| ! |  | Right-to-left |
| ( ) | Parentheses | Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses “on the same level” (i.e., not nested), they are evaluated left to right. |
| \* , / , or % | Multiplication Division Modulus | Evaluated second. If there are several, they are evaluated left to right. |
| + , - | Addition Subtraction | Evaluated last. If there are several, they are evaluated left to right. |
| Relational | < > <= >= | Evaluated last. If there are several, they are evaluated left to right |
| Equality | ==, != | Evaluated last. If there are several, they are evaluated left to right |
| And | && | Evaluated last. If there are several, they are evaluated left to right |
| OR | || | Evaluated last. If there are several, they are evaluated left to right |
| Assignment-level expressions | = \*= /= %= += -=>>= <<= &= ^= |= | Right-to-left |
| ++,-- postfix |  |  |

**Typecasting**

Whenever an integer and a floating point variable or constant are mixed in an operation, the integer is changed temporarily to its equivalent floating point. This automatic conversion is called implicit type conversion. Consider the following:

int count;

count = 7.8;

We are trying to put a floating point number into an integer memory location. This is like trying to stuff a package into a mailbox that is only large enough to contain letters. Something has to give. In C++ the floating point is truncated (the entire fractional component is cut off) and, thus, we have loss of information. Type conversions can be made explicit (by the programmer) by using the type casting or type conversion.

C++ supports the typecasting feature (inherited from C) to allow you to explicitly convert a value from one data type into another type. The general syntax for typecasting is:

// form 1

(*newType*)*expression*

// form 2

*newType*(*expression*)

Here are examples of using the typecasting feature:

char cLetter = 'A'

int nASCII = int(cLetter);

long lASCII = (long)cLetter;

This code snippet declares and initializes the **char**-type variable **cLetter**. The code also declares the **int**-type variable **nASCII** and initializes it using the **int** typecast of variable **cLetter**. In addition, the code declares the **long**-type variable **lASCII** and initializes it using the **long** typecast of variable **cLetter**.

**Example:**

int x,y;

float z = (float) x / y;// x is casting from int to float

or

float z = x / (float) y; // y is casting from int to float

or (unnecessary)

float z = (float) x / (float) y;

**Questions**

 **(SAMS - Object-Oriented Programming in C++ Book-chapter 2)**

**2, 3, 4, 5, 6, 9, 10, 11, 12, 14, 15, 17, 18, 19, 20, 21, 22, 23.**

2. A function name must be followed by \_\_\_\_\_\_\_\_.

3. A function body is delimited by \_\_\_\_\_\_\_\_.

4. Why is the main() function special?

5. A C++ instruction that tells the computer to do something is called a \_\_\_\_\_\_\_\_.

6. Write an example of a normal C++ comment and an example of an old-fashioned /\* comment.

9. True or false: A variable of type char can hold the value 301.

10. What kind of program elements are the following?

a. 12

b. ‘a’

c. 4.28915

d. JungleJim

e. JungleJim()

11. Write statements that display on the screen

a. the character ‘x’

b. the name *jim*

c. the number 509

12. True or false: In an assignment statement, the value on the left of the equal sign is always equal to the value on the right.

14. What header file must you #include with your source file to use cout and cin?

15. Write a statement that gets a numerical value from the keyboard and places it in the variable temp.

17. Two exceptions to the rule that the compiler ignores whitespace are \_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_.

18. True or false: It’s perfectly all right to use variables of different data types in the same arithmetic expression.

19. The expression 11%3 evaluates to \_\_\_\_\_\_\_\_.

20. An arithmetic assignment operator combines the effect of what two operators?

21. Write a statement that uses an arithmetic assignment operator to increase the value of the variable temp by 23. Write the same statement without the arithmetic assignment operator.

22. The increment operator increases the value of a variable by how much?

23. Assuming var1 starts with the value 20, what will the following code fragment print out?

cout << var1--;

cout << ++var1;

**Exercises**

1. Write a single C++ statement or line that accomplishes each of the following:

 a) Print the message "**Enter two numbers**".

 b) Assign the product of variables **b** and **c** to variable **a**.

 c) State that a program performs a sample payroll calculation (i.e., use text that helps to document a program).

d) Input three integer values from the keyboard and into integer variables **a**, **b** and **c**.

2. State which of the following are true and which are false. If false, explain your answers.

 a) C++ operators are evaluated from left to right.

b) The following are all valid variable names: **\_under\_bar\_, m928134, t5, j7, her\_sales, his\_account\_total, a, b, c, z, z2.**

c) The statement **cout << "a = 5;";** is a typical example of an assignment statement.

d) A valid C++ arithmetic expression with no parentheses is evaluated from left to right.

 e) The following are all invalid variable names: **3g, 87,**

 **67h2, h22, 2h**.

3. Fill in the blanks in each of the following:

a) What arithmetic operations are on the same level of precedence as multiplication?\_\_\_\_\_\_\_\_\_\_.

 b) When parentheses are nested, which set of parentheses is evaluated first in an arithmetic expression?\_\_\_\_\_\_\_\_\_\_.

 c) A location in the computer's memory that may contain different values at various times throughout the execution of a program is called a\_\_\_\_\_\_\_\_\_\_.

4. What, if anything, prints when each of the following C++ statements is performed? If nothing prints, then answer “nothing.” Assume x = 2 and y = 3.

 a) cout << x;

 b) cout << x + x;

 c) cout << "x=";

 d) cout << "x = " << x;

 e) cout << x + y << " = " << y + x;

 f) z = x + y;

 g) cin >> x >> y;

 h) // cout << "x + y = " << x + y;

 i) cout << "\n";

5. Which of the following C++ statements contain variables whose values are replaced?

 a) cin >> b >> c >> d >> e >> f;

 b) p = i + j + k + 7;

 c) cout << "variables whose values are replaced";

 d) cout << "a = 5";

6. Given the algebraic equation y = ax3 + 7, which of the following, if any, are correct C++ statements for this equation?

 a) y = a \* x \* x \* x + 7;

 b) y = a \* x \* x \* ( x + 7 );

 c) y = ( a \* x ) \* x \* ( x + 7 );

 d) y = (a \* x) \* x \* x + 7;

 e) y = a \* ( x \* x \* x ) + 7;

 f) y = a \* x \* ( x \* x + 7 );

7. State the order of evaluation of the operators in each of the following C++ statements and show the value of x after each statement is performed.

a) x = 7 + 3 \* 6 / 2 - 1;

b) x = 2 % 2 + 2 \* 2 - 2 / 2;

c) x = ( 3 \* 9 \* ( 3 + ( 9 \* 3 / ( 3 ) ) ) );

8. Write a program that asks the user to enter two numbers, obtains the two numbers from the user and prints the sum, product, difference, and quotient of the two numbers.

9. Write a program that prints the numbers 1 to 4 on the same line with each pair of adjacent numbers separated by one space. Write the program using the following methods:

a) Using one output statement with one stream insertion operator.

b) Using one output statement with four stream insertion operators.

c) Using four output statements.

10. Write a program that reads in the radius of a circle and prints the circle’s diameter, circumference and area. Use the constant value 3.14159 for π. Do these calculations in output statements.

11. Write a program to read the degree in Celsius and print it in Fahrenheit. C=( (5/9)\*(f-32)).

12. Write a program to read the length of a cube and print the surface area=6\*L2, the volume=L3, and the radius of the sphere= sqrt(3)\*L/2. (use math.h).

13. Write a program to ask the user to enter the speed in kilo meter and time in hour, then display the speed in the following forms:

* 1. Speed in km/h.
	2. Speed in km/m.
	3. Speed in meter per second.
	4. Speed in meter per minute.

14. Write an algorithm and a program that will read a number that represents the number of kilometers traveled. The output will convert this number to miles. 1 kilometer = 0.621 miles.

15. What does the following code print?

 cout << "\*\n\*\*\n\*\*\*\n\*\*\*\*\n\*\*\*\*\*\n";

16. Write a program that generates the following table:

1990 135

1991 7290

1992 11300

1993 16200

Use a single cout statement for all output.

17. Write a program that generates the following output:

 10

 20

 19

 Use an integer constant for the 10, an arithmetic assignment operator to generate the 20, and a decrement operator to generate the 19.

18. Write a program that accept X, Y for two points P1 = (X1, Y1) and P2 = (X2, Y2) and calculate the Slope (m) = X1-X2 / Y1-2Y.

19. Write a program to calculate Total Resistance of three resistances in parallel. Rt = R1 \*R2 \* R3/ R1+\*R2 + R3.

20. Write a program to calculate Total Resistance of four resistances in circuit, three of them connected in parallel and the last one connected in series with them. Rt = R in parallel + R in series.

21. Write a program to calculate I total (It) of a circuit contains voltage source (Vs) and five resistances connected in parallel. It = Vs / Rt

**The Simple if Statement**

C++ offers the simple **if** statement to support single-alternative decision making. The general syntax for the simple **if** statement is:

// form 1

if (*condition*)

  *statement*;

// form 2

if (*condition*) {

  // sequence of statement

}

Here are examples of the single-alternative **if** statement:

// example 1

if (nNum < 0)

  cout << "Value is negative!\n";

 // example 2

if (i > 0 && i < 100)

  cout << "Number is in range 1 to 99\n";

// example 3

if (nCount < 1)

  nCount = 1;

**The if-else Statement**

C++ enables the **if** statement to support dual-alternative decision making. The general syntax for the dual-alternative **if** statement is:

if (*condition*)

  // statement or block of statements

else

  // statement or block of statements

Here are examples of the dual-alternative **if** statement:

// example 1

if (nNum < 0)

  cout << "Value is negative\n";

else

  cout << "Value is 0 or greater\n"

// example 2

if (i > 0 && i < 100)

  j = i \* i;

else

   j = 100;

// example 3

if (nCount < 1)

  nCount = 1

else

  nCount--;

**The Multiple-Alternative if Statement**

C++ also permits the **if** statement to support multiple-alternative decision making. The general syntax for the multiple-alternative **if** statement is:

if (*condition1*)

  // statement #1 or block of statements #1

else if (*condition2*)

  // statement #2 or block of statements #2

else if (*condition3*)

  // statement #3 or block of statements #3

// other else if clauses

else

  // catch-all statement or catch-all block of statements

Here is an example of a multiple-alternative **if** statement:

if (N >= 0 && N < 10)

  cout << "Variable N is a single digit\n";

else if (N >= 10 && N < 100)

  cout << "Variable N has two digits\n";

else if (N >= 100 && N < 1000)

  cout << "Variable N has three digits\n";

else if (N >= 1000)

  cout << "Variable N has four or more digits\n";

else

  cout << "Variable N is negative\n";

**Example:**

1-Write a C++ program that accepts grade for a student then determines his graduation grade according to the incoming table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Criteria | g > =90 | 90>g>=80 | 80>g>=70 | 70>g>=60 | 60>g |
| Income level | Excellent | Very good | good | Pass | Fail |

Interaction with the program might look like this:

Enter a grade: 85

Graduation grade: very good

#include <iostream.h>

int main()

{ int grade;

 cout<<"Enter grade : ";

 cin>>grade ;

 if(grade >=90)

 cout<<"\n Excellent"<<endl;

 else if (grade>=80)

 cout<<"\n Very good"<<endl;

 else if (grade>=70)

 cout<<"\n good"<<endl;

 else if (grade>=60)

 cout<<"\n pass"<<endl;

 else

 cout<<"\n Fail"<<endl;

 return 0;

}

2- Write a program to get two numbers and the arithmetic operation, and print the result.

3- Write a program to solve ax2+bx+c=0.

**The switch Statement**

C++ offers the **switch** statement to support multiple-alternative decision making. The general syntax for the multiple-alternative **switch** statement is:

switch(expression)

{

  case *constantExpression1*:[case constExp6:]

    // statement set #1

    break;

 case *constantExpression2*:

    // statement set #2

    break;

. . .

[default:]

  // catch-all statements*]*

}

The **switch** statement examines the value of the expression, which must be a value or function output with type integer or character.

**Example:**

1-Write a program that asks the user to enter the item number and

 print the price of this item according to the following table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ItemNo** | 1 | 2 | 3 | 4 |
| **Price** | 100 | 200 | 600 | 150 |

#include <iostream.h>

int main()

{

 int ItemNo;

 cout<<"Enter the item no. : ";

 cin>>ItemNo ;

 switch(ItemNo)

 {

 case 1:

 cout<<"\n the price is 100"<<endl;

 break;

 case 2:

 cout<<"\n the price is 200"<<endl;

 break;

 case 3:

 cout<<"\n the price is 600"<<endl;

 break;

 case 4:

 cout<<"\n the price is 150"<<endl;

 break;

 default:

 cout<<"\n the item no. not found"<<endl;

 }

return 0;

}

2- Write a program that reads your birthday and the current date, and prints your age.

#include <iostream.h>

int main()

{ int urd, urm,ury,cd,cm,cy,rd,rm,ry;

 cout<<"Enter your birthday"<<endl;

 cin>>urd>>urm>>ury;

 cout<<"Enter the current date"<<endl;

 cin>>cd>>cm>>cy;

 if(urd>cd)

 {

 switch(cm)

 {

 case 4:case 6:case 9:case 11: cd+=30;

 break;

 case 1:case 3:case 5:case 7: case 8: case 10: case

 12:cd+=31;

 break;

 case 2: if (cy%4==0)

 cd+=29;

 else

 cd+=28;

 break;

 }

 cm--;

 }

 if(urm>cm)

 {

 cm+=12; cy--;

 }

 rd=cd-urd;

 rm=cm-urm;

 ry=cy-ury;

 cout<<"your age is "<<rd<<":"<<

 rm<<":"<<ry<<endl;

return 0;

}

**Exercises**

1. Write a program that asks the user to enter two integers, obtains the numbers from the user, then prints the larger number followed by the words "is larger." If the numbers are equal, print the message “These numbers are equal.”

2. Write a program that inputs three integers from the keyboard and prints the sum, average, product, smallest and largest of these numbers. The screen dialogue should appear as follows:

Input three different integers: 13 27 14

Sum is 54

Average is 18

Product is 4914

Smallest is 13

Largest is 27

3. Write a program that reads in five integers and determines and prints the largest and the smallest integers in the group. Use only the programming techniques you learned in this lecture.

4. Write a program that reads an integer and determines and prints whether it is odd or even. (Hint: Use the modulus operator. An even number is a multiple of two. Any multiple of two leaves a remainder of zero when divided by 2.)

5. Write a program that reads in two integers and determines and prints if the first is a multiple of the second. (Hint: Use the modulus operator.)

6. Write an algorithm and a C++ program that accepts an employee salary and a tax percentage then computes the monthly net salary and determine his income level according to the incoming table:

|  |  |  |  |
| --- | --- | --- | --- |
| Criteria | ns > =5000 | 50000>ns>=2000 | ns<2000 |
| Income level | High | moderate | low |

**Loops**

**For Loop**

The general syntax for the **for** loop is:

for (*initilizationPart* ; *ConditionPart* ; *incrementPart*)

{

 Body;

}

The initialization part, which initializes the loop control variable(s). You can use single or multiple loop control variables.

2. The iteration part, which contains a Boolean expression that causes the loop to iterate as long as the expression is true

3. The increment part, which increments or decrements the loop control variable(s)

Here are examples of the **for** loop:

// example 1

for (i = 0; i < 10; i++)

  cout << i << "\n";

// example 2

for (i = 9; i >= 0; i -= 3)

  cout << (i\*i) << "\n";

// example 3

for (int i = 1; i < 100; i++)

  cout << i << "\n";

// example 4

for (int i = 0, j = MAX; i < j; i++, j--)

  cout << (i + 2 \* j) << "\n";

**Example**

1. Write a program to print the values from 1-10.
2. Write a program to print the even values from 1-10.
3. Write a program to print the count, sum, and average of the even values from 1-100.
4. Write a program to read a value and print the factorial.
5. Write a program to read a value and determine if it is a prime or not.
6. Write a program to read a value and determine if it is a perfect or not.
7. Write a program to convert from decimal to binary.

**An open-iteration For loop**

int i = 0; // initialize variable

int j;

for (;;)

 {

  cout << i << " : ";

// test loop condition and exit if i >= 10

  if (i >= 10)

    break;

  j = i + i \* i - 5;

  i++; // increment loop control variable

  cout << j << "\n";

}

**While Loop**

The syntax for the **while** loop is:

*InitializationPart*

while (*conditionPart*)

{

// statement or statement block

*IncrementPart/decrementPart;*

}

The syntax of the **while** loop shows that it tests the iteration condition *before* executing the loop's statement. Thus, the **while** loop will not execute if the tested condition is already false. Here is an example of the **while** loop:

int i = 0;

while (i \* i < 1000)

  i++;

This example has a **while** loop that iterates as long as the squared value of variable **i** is less than 1000.

**Example**

The following program finds and prints the smallest integer that is able to divide by 12 and 14

#include <iostream.h>

 int main()

 {

 int i=1;

 while(i%12!=0 || i%14!=0)

 i++;

 cout <<i<<endl;

 return 0;

 }

**Do-while Loop**

The **do-while** loop iterates as long as a tested condition is true. The syntax for the **do-while** loop is:

*InitializationPart;*

do {

  // statements

*Increment/decrement part;*

} while (*conditionPart*);

The syntax of the **do-while** loop shows that: it tests the iteration condition *after* executing the loop's statement. Thus, the **do-while** loop always executes at least once. Here is an example of the **do-while** loop:

do{

  cout << "Enter a positive integer : ";

  cin >> nNum;

}while (nNum < 1);

This example shows a **do-while** loop that iterates as long as the value in the variable **nNum** is less than 1.

**Exiting Loops**

The **break** statement exits the current loop. Thus to exit nested loops, you need to use a **break** statement for each loop.

Here are examples of using the **break** statement with the **do-while** and **while** loops:

// exit from do-while loop example

double fY, fX = 1.0;

do {

  fY = fX \* fX + 10;

  if (fY > 10000.0)

     break;

  cout << "f(" << fX << ") = " << fY << "\n";

fX+=10;

} while (fX < 100.0);

// exit from while loop example

double fY, fX = 1.0;

while (fX > 0.0 && fX < 100.0) {

  fY = fX \* fX - 30;

  cout << "f(" << fX << ") = " << fY << "\n";

  if (fY > 1000.0)

     break;

fX+=10;

}

**Skipping Loop Iterations**

C++ offers the **continue** statement to skip the remaining statements in a loop. Why skip the remaining loop statements? This condition arises when the loop statements examine a condition and conclude that the loop should not or need not proceed with executing the remaining statements. Here is an example of using the **continue** statement:

for (int i = -4; i < 5; i++) {

  if (i == 0)

  continue;

  double fX =  1.0 / i;

  cout << "1 / " << i << " = " << fX << "\n";

}

This code snippet shows a loop that displays reciprocal values. The loop has a control variable that changes values from —4 to 4, in increments of 1. The loop contains an **if** statement that determines whether or not the control variable contains 0. When this condition is true, the loop skips the remaining statements to *avoid* dividing by zero!

**Nested Loops**

C++ allows you to nest loops in any combination. For example, you can nest **for** loops, as shown in the following code snippet:

double fSum = 0;

for (int i = 10; i < 100; i++)

  for (int j = 0; j < i; j++)

    fSum += double(j \* i);

This code snippet shows two nested **for** loops used to obtain a summation.

You can also nest different kinds of loops. Here is a code snippet that shows you nested **while** and **do-while** loops:

double fSum = 0;

int i = 10;

int j;

while (i < 100)

{

  j = 0;

   do {

      fSum += double(j++ \* i);

   } while  (j < i);

   i++;

}

The nested loops obtain a summation, like the one in the example of the nested **for** loops.

**Example**

1. Write a program to print the following pyramid.

 **\*
    \*\***

 **\*\*\*\***

 **\*\*\*\*\*\***

 **\*\*\*\*\*\*\*\***

 **\*\*\*\*\*\*\*\*\*\***

# include<iostream.h>

int main()

{

 for(int i=1;i<=5;i++)// for draw 5 lines

{

 for(int j=1;j<=5-i;j++)// for spaces

 cout<<" ";

 for(int k=1; k<=i;k++)

 cout<<"\* ";

 cout<<endl;

 }

}

1. Write a program to print the full pyramid.

# include<iostream.h>

int main()

{

 int l;

 for (int i=1;i<=5;i++)// for draw 5 lines

 {

 for(int j=1;j<=5-i;j++)// for spaces

 cout<<" ";

 for(int k=1; k<=i;k++)

 cout<<"\* ";

 cout<<endl;

 }// the half is printed

for (i=4;i>=1;i--)// for draw 5 lines

 {

 for(int j=1;j<=5-i;j++)// for spaces

 cout<<" ";

 for(int k=1; k<=i;k++)

 cout<<"\* ";

 cout<<endl;

 }// the full is printed

}

1. Write a program to draw the following shape

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*

\*\*\*

\*

\*

\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

# include<iostream.h>

int main()

{ for (int i=5;i>=1;i--)// for draw 5 lines

 {for(int j=1;j<=2\*i-1;j++)// for spaces

 cout<<"\*";

 cout<<endl;

 }// the half is printed

for (i=1;i<=5;i++)// for draw 5 lines

{

 for(int j=1;j<=i;j++)// for spaces

 cout<<"\*";

 cout<<endl;

 }// the full is printed

}

See the following link

<http://www.programiz.com/article/c%2B%2B-programming-pattern#reverse_pyramid>

1. Write a program to draw an alphabetic pyramid.

# include<iostream.h>

int main()

{ char x='A';

 cout<<" "<<x<<endl;

 for (int i=1;i<=4;i++)// for draw 5 lines

 {

 for(int j=1;j<=4-i;j++)// for spaces

 cout<<" ";

 cout<<++x;

 for(int k=1; k<=2\*i-1;k++)

 cout<<"-";

 cout<<x;

 cout<<endl;

 }// the half is printed

for (i=3;i>=1;i--)// for draw 5 lines

{

 for(int j=1;j<=4-i;j++)// for spaces

 cout<<" ";

 cout<<--x;

 for(int k=1; k<=2\*i-1;k++)

 cout<<"-";

 cout<<x;

 cout<<endl;

 }// the full is printed

 cout<<" "<<--x<<endl;

}

**Exercises**

1. Identify and correct the error(s) in each of the following:

 a) if ( age >= 65 );

 cout << "Age is greater than or equal to 65" << endl;

else

 cout << "Age is less than 65 << endl";

 b) if ( age >= 65 )

 cout << "Age is greater than or equal to 65" << endl;

else;

 cout << "Age is less than 65 << endl";

 c) int x = 1, total;

while ( x <= 10 ) {

 total += x;

 ++x;

}

 d) While ( x <= 100 )

 total += x;

 ++x;

}

 e) while ( y > 0 ) {

 cout << y << endl;

 ++y;

}

|  |
| --- |
| 2.What does the following program print? |

|  |  |  |
| --- | --- | --- |
| space50x1 |

|  |
| --- |
|   1)   #include  <iostream>  2)     3)   using  std::cout;  4)   using  std::endl;  5)     6)   int  main()  7)   {  8)       int  y, x = 1, total = 0;  9)    10)       while  ( x <= 10 ) { 11)         y = x \* x; 12)         cout << y << endl; 13)         total += y; 14)         ++x; 15)      } 16)    17)     cout << "Total is " << total << endl; 18)       return  0; 19)   } |

 |

3. Write a C++ program that utilizes looping and the tab escape sequence \t to print the following table of values:

|  |  |  |
| --- | --- | --- |
| space50x1 |

|  |
| --- |
| N       10\*N    100\*N   1000\*N     1       10      100     10002       20      200     20003       30      300     30004       40      400     40005       50      500     5000 |

 |

4. What does the following program print?

|  |
| --- |
|   1)   #include <iostream>  2)     3)   using std::cout;  4)   using std::endl;  5)     6)   int main()  7)   {  8)      int count = 1;  9)    10)      while ( count <= 10 ) { 11)         cout << (count % 2 ? "\*\*\*\*" : "++++++++")  12)              << endl; 13)         ++count; 14)      } 15)    16)      return 0; 17)   }  |

5. What does the following program print?

|  |
| --- |
|   1)   #include <iostream>  2)     3)   using  std::cout;  4)   using  std::endl;  5)     6)   int  main()  7)   {  8)       int  row = 10, column;  9)    10)       while  ( row >= 1 ) { 11)         column = 1; 12)    13)          while  ( column <= 10 ) { 14)            cout << (row % 2 ? "<" : ">"); 15)            ++column; 16)         } 17)    18)         --row; 19)         cout << endl; 20)      } 21)    22)       return  0; 23)   }  |

6. Determine the output for each of the following when **x** is **9** and **y** is **11** and when **x** is **11** and **y** is **9**.

a) if ( x < 10 )

if ( y > 10 )

cout << "\*\*\*\*\*" << endl;

else

cout << "#####" << endl;

cout << "$$$$$" << endl;

 b) if ( x < 10 ) {

if ( y > 10 )

cout << "\*\*\*\*\*" << endl;

}

else {

cout << "#####" << endl;

cout << "$$$$$" << endl;

}

7. Modify the following code to produce the output shown. Use proper indentation techniques. You must not make any changes other than inserting braces. Note: It is possible that no modification is necessary.

 if ( y == 8 )

 if ( x == 5 )

 cout << "@@@@@" << endl;

 else

 cout << "#####" << endl;

 cout << "$$$$$" << endl;

 cout << "&&&&&" << endl;

 a) Assuming x = 5 and y = 8 , the following output is produced.

 @@@@@

$$$$$

&&&&&

 b) Assuming x = 5 and y = 8 , the following output is produced.

@@@@@

c) Assuming x = 5 and y = 8 , the following output is produced.

@@@@@

&&&&&

d) Assuming x = 5 and y = 7 , the following output is produced. Note: The last three output statements after the else are all part of a compound statement.

#####

$$$$$

&&&&&8. Write a program that reads in the size of the side of a square and then prints a hollow square of that size out of asterisks and blanks. Your program should work for squares of all side sizes between 1 and 20. For example, if your program reads a size of 5, it should print

\*\*\*\*\*

\* \*

\* \*

\* \*

\*\*\*\*\*

9. Write a program that displays the following checkerboard pattern

\* \* \* \* \* \* \* \*

 \* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \*

 \* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \*

 \* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \*

 \* \* \* \* \* \* \* \*

 Your program must use only three output statements, one of each of the following forms:

 cout << "\* ";

cout << ' ';

cout << endl;

10. Write a program that reads three nonzero double values and determines and prints if they could represent the sides of a triangle.

11. Write a program that reads three nonzero integers and determines and prints if they could be the sides of a right triangle.

12. Find the error(s) in each of the following:

 a) For ( x = 100, x >= 1, x++ )

 cout << x << endl;

 b) The following code should print whether integer value is odd or even:

switch ( value % 2 )

 case 0:

 cout << "Even integer" << endl;

 case 1:

 cout << "Odd integer" << endl;

}

 c) The following code should output the odd integers from 19 to 1:

for ( x = 19; x >= 1; x += 2 )

 cout << x << endl;

 d) The following code should output the even integers from 2 to 100:

counter = 2;

do {

 cout << counter << endl;

 counter += 2; }

While ( counter < 100 );

13. Write a program that finds the smallest of several integers. Assume that the first value read specifies the number of values remaining and that the first number is not one of the integers to compare.

14 Write a program that calculates and prints the product of the odd integers from 1 to 15.

15. Write a program that prints the following patterns separately one below the other. Use for loops to generate the patterns. All asterisks (\*) should be printed by a single statement of the form cout << '\*';

**(A) (B) (C) (D)**

**\* \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* \***

**\*\* \*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\* \*\***

**\*\*\* \*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\* \*\*\***

**\*\*\*\* \*\*\*\*\*\*\* \*\*\*\*\*\*\* \*\*\*\***

**\*\*\*\*\* \*\*\*\*\*\* \*\*\*\*\*\* \*\*\*\*\***

**\*\*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\*\***

**\*\*\*\*\*\*\* \*\*\*\* \*\*\*\* \*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\* \*\*\* \*\*\* \*\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\*\* \*\* \*\* \*\*\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\*\*\* \* \* \*\*\*\*\*\*\*\*\*\***

16. Assume i = 1, j = 2, k = 3 and m = 2. What does each of the following statements print? Are the parentheses necessary in each case?

 a) cout << ( i == 1 ) << endl;

 b) cout << ( j == 3 ) << endl;

 c) cout << ( i >= 1 && j < 4 ) << endl;

 d) cout << ( m <= 99 && k < m ) << endl;

 e) cout << ( j >= i || k == m ) << endl;

 f) cout << ( k + m < j || 3 - j >= k ) << endl;

 g) cout << ( !m ) << endl;

 h) cout << ( !( j - m ) ) << endl;

 i) cout << ( !( k > m ) ) << endl;

**Arrays**

**Single-Dimensional Arrays**

It’s used to store many values as a group using one name and different indices. The general syntax for declaring an array is:

**Type Name [length] ;**

The **length** is a **constant integer number and must be greater than zero**. The **name** must apply the naming convention rules. In C++, indices start from 0 to length-1. Thus, the number of array elements is one larger than the index of the last array element. **Here are examples of declaring arrays:**

***//* example 1**

int IntArr[l0];

int IntArr[0]; // syntax error

***//* example 2**

const int MAX = 30;

char cName[MAX]; // string of length 29

***//* example 3**

const int MAX\_CHARS = 40;

char cString[MAX \_CHARS+ 1]; // string of length 40

**Initializing Single-Dimensional Arrays**

Type name [length] = {v\_1,v\_2,….v\_length-1};

1-The list of initial values appears in a pair of open and close braces and is comma-delimited. The list ends with a semicolon.

2. The list may contain a number of initial values that is equal to or less than the number of elements in the initialized array. Otherwise, the compiler generates a compiler-time error.

3. The compiler assigns the first initializing value to the element at index 0, the second initializing value to the element at index 1, and so on.

4. If the list contains fewer values than the number of elements in the array, the compiler assigns zeros to the numeric elements and ‘\0’ (null) to the character elements that do not receive initial values from the list.

5. In initializing a character array (with or without length), the last cell is ‘\0’ to specify the end of the array. Therefore, if you want to assign a name of size 4, declare an array of size 5.

 6. If you omit the number of array elements, the compiler uses the number of initializing values in the list as the number of array elements.

Here are examples of initializing arrays:

*//* **example 1**

double fArr[5] = { 1.1,2.2,3.3,4.4,5.5 };

*//* **example 2**

int nArr[10] = { 1,2,3,4,5 };// the other cells are 0

// **example 3**

char name[4]="zizo"; //error, because the size must be 5 for inserting ‘\0’ automatically.

char name[5]="zizo"; //right, automatically insert ‘\0’, and cout<<name prints a correct value “zizo”;

char name[]=”zizo”; //right, the size is 5, automatically insert ‘\0’, and cout<<name prints a correct value “zizo”;

// example 4

Char name[4]={‘z’,’I’,’z’,’o’};// the size will be 5 automatically to insert ‘\0’ and cout<<name will print extra characters.

Char name[5]={‘z’,’I’,’z’,’o’}// the size is 5, ‘\0’ will be inserted automatically, and cout<<name will print a correct name “zizo”.

Char name[]={‘z’,’I’,’z’,’o’}// the size will be 5, ‘\0’ will be inserted automatically and cout<<name will print extra characters.

Char name[]={‘z’,’I’,’z’,’o’,’\0’}// the size is 5, ‘\0’ is inserted manually, and cout<<name will print a correct name “zizo”.

In both cases, the array of characters name is declared with a size of 5 elements of type char: the 4 characters that compose the word "zizo", 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.

**Reading and Printing the Array**

int values[10];

for (int i=0;i<10;i++)

cin>>values[i];

for (int i=0;i<10;i++)

cout<<values[i]<<endl;

char name[4];

cin>>name;

cout<<name;

// 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; // no space

 cout << greeting << yourname << "!"; return 0;}

**Example**

1. Write a program to read 100 values and print the average of them.
2. Write a program to read 100 values and print the average of the odd values.
3. Write a program to read 100 values and print the average of the values which are multiple of 3.
4. Write a program to read 100 values and find the max and min.
5. Write a program to read a name and print it in reverse order.
6. Write a program to read 10 values and print them in an ascending order (sort).
7. Write a program to read 10 values and another value. The program prints the index of the value if it finds it.

 **Multidimensional Arrays**

**Declaring Multidimensional Arrays**

The general syntax for declaring a multidimensional array is:

Type arrayName[ row][column];

All multidimensional arrays in c++ have indices that start at 0. Thus, the number of array elements in each dimension is one value higher than the index of the last element in that dimension.

*//* **example 1**

int nlntCube[20][1 0][5];// 3 dimensional

*//* **example 2**

const int MAX\_ROWS = 50;

const int MAX\_ COLS = 20;

double fMatrix[MAX\_ROWS][MAX\_COLS];

const int MAX\_ROWS = 30;

const int MAX\_COLS = 10;

char cNarneArray[MAX\_ ROWS+ I][MAX \_COLS];

**Initializing Multidimensional Arrays**

Type arrayName [numberOfElementl][numberOfElement2] = {.valueO ,... ,valueN };

1-The list of initial values appears in a pair of open and close braces and is comma-delimited.

2. The list may contain a number of initial values that is equal to or less than the total number of elements in the initialized array. Otherwise, the compiler generates a compile-time error.

3. The compiler assigns the initializing values in the sequence discussed in the sidebar "Initializing Multidimensional Arrays."

4. If the list contains fewer values than the number of elements in the array, the compiler assigns zeros to the numeric elements and ‘\0’ (null) to the character elements that do not receive initial values from the list.

Here are examples of initializing multidimensional arrays:

*//* **example 1**

double fMat[2][3] = {1.1,2.2,3.3,4.4,5.5,6.6}; // row1 is //1.1,2.2,3.3 and row2 is 4.4,5.5,6.6

*//* **example 2**

int nMat[2][5] = { 1,2,3,4,5 }; //row2 is empty

*//* **example 3**

double fMat[3][2] = {{ 1.1, 2.2}, {3.3, 4.4}, {5.5, 6.6} };

**Reading and Printing the Multidimensional Arrays**

const int MAX\_ROWS = 10;

const int MAX\_COLS = 20;

double fMatrix[MAX\_ROWS][MAX \_COLS];

for (int i = 0; i < MAX ROWS; i++)

for (intj = 0;j <MAX\_COLS;j++)

fMatrix[i][j] = double(2 + ie \* j);

**Example**

1. Write a program to read two matrices and print their sum.
2. Write a program to read two matrices and print their multiply.

**Functions**

**Functions definition**

All C++ functions have certain basic features. Each function has a **name**, a **return type**, and an **optional parameter list**. Functions can declare local constants and variables. Except for the function main (), you should prototype functions (that is declare them in advance). C++ functions have the following syntax:

returnType functionName (parameterList)

{

 // declarations

 // statements

 return expression;

}

Every function has a **return type** that appears before the name of the function. The **parameter list** follows the function's name and is enclosed in parentheses. The function returns a value using the **return** statement that typically appears at the end. A function may have more than one return statement.

The parameter list of a function may contain one or more parameters, which correspond to the arguments given the function when it is actually called. The list of parameters is comma-delimited, and each parameter has the following syntax:

parameterType [&] parameterName

You need to observe the following rules about the parameters of a function:

* Each parameter must have its own type. You cannot use the same type to declare multiple parameters (as you can when declaring variables).
* If a function has no parameters, the parentheses that come after the function's name contain nothing.
* The argument for a parameter is passed by copy (or, as it is sometimes said, “**by value**”), unless you insert the **reference-of operator** & after the parameter's type. When a parameter passes a copy of its argument, the function can alter only the copy of the argument used within the function itself. The original argument remains intact. By contrast, using the reference-of operator allows the argument to be passed by reference by declaring the parameter as a reference to its argument. In this case, the parameter becomes a special alias to its argument. Any changes the function makes to the parameter also affect the argument.
* Reference parameters take arguments that are the names of variables. You cannot use an expression or a constant as an argument to a reference parameter since an expression does not have an address as a variable does.
* Copy parameters take arguments that are constants, variables, or expressions. The type of argument must either match the type of the parameter or be compatible with it. You may use typecasting to tell the compiler how to adjust the type of the argument to match the type of the parameter.

Here are examples of functions:

double getSquare(double x) // one parameter

{

 return x \* x;

}

double Square(double& x) // one parameter, modifies its argument

{

 x = x \* x;

 return x;

}

int getMin(int nNum1, int nNum2) // two parameters

{

 return (nNum1 < nNum2) ? nNum1 : nNum2;

}

int getSmall(int nNum1, int nNum2, int nNum3); // three parameters

{

 if (nNum1 < nNum2 && nNum1 < nNum3)

 return nNum1;

 else if (nNum2 < nNum1 && nNum2 < nNum3)

 return nNum2;

 else

 return nNum3;

}

The first function, getSquare, has the return type double and the single double-type parameter x. The function returns the squared value of the parameter x. The function getSquare contains a single statement, namely the return statement.

The second function, Square, has the return type double and the single double-type reference parameter x. The function squares the value of the reference parameter and returns the new value in x (this value also affects the argument for function Square). Therefore, the function returns the squared value in two ways: first as the function's return value, and second using the reference parameter x.

The third function, getMin, has the return type int and the two int-type parameters nNum1 and nNum2. The function returns the smaller of the values supplied by the arguments for the parameters. The function getMin has a single statement that returns the minimum number sought. This statement uses the conditional assignment operator.

The fourth function, getMin, has the return type int and the three int-type parameters nNum1, nNum2, and nNum3. The function returns the smallest value supplied by the arguments for the three parameters. The function getSmall uses a multiple-alternative if-else statement to obtain the sought-after minimum.

**The Function Declaration**

C++ also supports the forward declaration of functions (which is called prototyping). The forward declaration allows you to list the functions at the beginning of the source code. Such a list offers a convenient way to know what functions are in a source code file. In addition, using the prototypes gives the compiler advance notice of the names, return types, and parameter lists of the various functions. You can then place the definitions of the functions in any order and not worry about the compile-time errors that occur when you call a function before you either declare it or define it. The general syntax for a function prototype is:

returnType functionName(parameterList);

Notice that the semicolon at the end is needed for the prototype but does not work in the function definition.

Here are the function declaration for the functions that are presented in the last section:

double getSquare(double x); // one parameter

double Square(double& x); // one parameter

int getMin(int nNum1, int nNum2); // two parameters

int getSmall(int nNum1, int nNum2, int nNum3); // three parameters

**Example**:

1. Write a C++ program that computes and prints the summation of the even numbers between 20 and 30. This program uses function to determine if the number is even or not.

#include<iostream.h>

bool even(int x);

int main()

{

 int sum=0;

 for (int i=20;i<=40;i++)

 {

 if(even(i))

 sum+=i;

 }

 cout<<"sum= "<<sum<<endl;

 return 0;

}

bool even(int x)

{

 if (x%2==0)

 return true;

 else

 return false;

}

**Math Library Functions**

Math library functions allow the programmer to perform certain common mathematical calculations. All functions in the math library return the data type double. To use the math library functions, include the header file <cmath>. Some math library functions are summarized in the following table. In the table, the variables x and y are of type double.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|

|  |  |  |
| --- | --- | --- |
| Method | Description | Example |
| **ceil( x )** | rounds x to the smallest integer not less than x | ceil( 9.2 ) is **10.0** ceil( -9.8 ) is -9.0 |
| **cos( x )** | trigonometric cosine of x (x in radians) | cos( 0.0 ) is 1.0 |
| **exp( x )** | exponential function ex | exp( 1.0 ) is 2.71828 exp( 2.0 ) is 7.38906 |
| **fabs( x )** | absolute value of x  | fabs( 5.1 ) is 5.1 fabs( 0.0 ) is 0.0 fabs( -8.76 ) is **8.76** |
| **floor( x )** | rounds x to the largest integer not greater than x | floor( 9.2 ) is 9.0 floor( -9.8 ) is -10.0 |
| **fmod( x, y )** | remainder of x/y as a floating point number | fmod( 13.657, 2.333 ) is 1.992 |
| **log( x )** | natural logarithm of x (base e) | log( 2.718282 ) is 1.0 log( 7.389056 ) is 2.0 |
| **log10( x )** | logarithm of x (base 10) | log10( 10.0 ) is 1.0 log10( 100.0 ) is 2.0 |
| **pow( x, y )** | x raised to power y (xy) | pow( 2, 7 ) is 128 pow( 9, .5 ) is 3 |
| **sin( x )** | trigonometric sine of x (x in radians) | sin( 0.0 ) is 0 |
| **sqrt( x )** | square root of x | sqrt( 900.0 ) is 30.0 sqrt( 9.0 ) is 3.0 |
| **tan( x )**  | trigonometric tangent of x (x in radians) | tan( 0.0 ) is 0 |

 |

**Recursive Functions**

Recursion is a method in which a function obtains its result by calling itself. Successive recursive calls must pass different arguments and must reach a limit or condition where the function stops calling itself. These two simple rules prevent a recursive function from indefinitely calling itself. Conceptually, recursion is a form of iteration that does not use the formal fixed or conditional loop. Many algorithms (such as calculating factorials and performing a quicksort) can be implemented using either recursive functions or straightforward loops. Some algorithms are easier to implement using recursion. An example is the algorithm for parsing and evaluating mathematical expressions. This is because an expression may contain smaller expressions and therefore, recursion offers the best solution. In other words, the main expression may contain nested expressions. Here is an example:

Z = ((X + Y) \* X)) + (X \* Y) / (1 + X);

The above statement contains the nexted expressions ((X + Y) \* X)), (X + Y), (X \* Y), and (1 + X).

**Example:**

The following function calculates factorials.

#include<iostream.h>

int factorial(int x);

 int main()

{

 int no;

 cin>>no;

 if(no<0)

 {

 cout<<" \n Invalid input"<<endl;

 return 0;

 }

 else

 cout<<" \n factorial= "<<factorial(no)<<endl;

 return 0;

}

int factorial(int x)

{

 if (x<=1) return 1;

 else

 return x\*(factorial(x-1));

}

**Default Arguments**

C++ allows you to assign default arguments for parameters. The syntax for the default argument is:

parameterType parameterName = initialValue

C++ requires that you observe the following rules for declaring and using default arguments:

1. When you assign a default argument to a parameter, you must assign default arguments to all subsequent parameters.
2. You may assign default arguments to any or all parameters, as long as you obey rule number 1.
3. The default arguments feature divides the parameter list of a function into two parts. The first part contains parameters with no default arguments (this list may be empty if you assign default arguments to all parameters); the second part contains parameters with default arguments.
4. To use a default argument for a parameter, omit the argument for that parameter in a function call.
5. If you use a default argument for a parameter, you must use default arguments for all subsequent parameters. In other words, you cannot pick and choose the default arguments, because the compiler is unable to discern which argument goes to which parameter. (After all, this is programming and not black magic!)

For example, the following declaration declares the myPower function:

double myPower(double fBase, int nExponent = 2);

The function myPower has the double-type parameter fBase and the int-type parameter nExponent. The latter parameter has the default argument of 2. Thus, you can use the function myPower in this fashion:

double fX = 12.5;

double fXSquared = myPower(fX);

double fXCubed = myPower(fX, 3);

The first call to function myPower has only one argument. The compiler resolves this call by using the default argument of 2 for parameter nExponent. Consequently, the function myPower returns the square of the first argument’s value when you omit the argument for the exponent. By contrast, the second call to function myPower uses the arguments fX and 3. In this case, the compiler does not use the default argument for parameter nExponent, since it has been given both arguments explicitly.

**Example**:

#include<iostream.h>

int sum(int x=2,int y=4,int z=3);

int main()

{

 int m;

 m=sum();

 cout<<"sum()="<<m<<endl;

 m=sum(1);

 cout<<"sum(1)="<<m<<endl;

 m=sum(5,6);

 cout<<"sum(5,6)="<<m<<endl;

 return 0;

}

int sum(int x,int y,int z)

{

 int n=x+y+z;

 return n;

}

**Constant Parameters**

By default, a function can alter the data passed by the arguments to its parameters. If the parameter is a not a reference parameter, then the changes made to the argument are limited to the function's scope. By contrast, if the parameter is a reference parameter, then the changes made to the argument go beyond the function's scope. You can tell the compiler that the function should not alter the argument of a parameter by declaring that parameter as a **constant parameter**. The declaration uses the keyword const and has the following general syntax:

const parameterType[&] parameterName [= defaultArg]

**Example**: The following example shows how to use pointers:

int x;

int\* px;

cin >> x;

px = &x;

\*px =\*px+ 10;

cout << \*px; // display value in variable nCount

**Example**: this example shows the call by reference:

#include<iostream.h>

void sum(int \*x,int\* y);

int main()

{

int \*n,\*m;

n=new int;

m=new int;

\*n=2;

\*m=3;

sum(n,m);

cout<<"sum= "<<\*n<<endl;

return 0;

}

void sum(int \*x ,int \*y)

{

 \*x=\*x+\*y;

 return;

}

To prevent the function from changing the argument we use const before the argument as follow:

#include<iostream.h>

int sum(const int \*x, const int\* y);

int main()

{

int \*n,\*m;

n=new int;

m=new int;

\*n=2;

\*m=3;

int p=sum(n,m);

 cout<<"sum= "<<p<<endl;

 return 0;

}

int sum(const int \*x, const int \*y)

{

 int z=\*x+\*y;

 return z;

}

**Function Overloading**

Overloaded functions in C++ are a valuable feature that allows you to declare functions in sets of versions that have the same name but different parameters. These parameters form each version’s signature. Using an overloaded function empowers you to use the same name for a set of function versions that perform similar tasks on different data types. For example, you can define the function Square to obtain the squares of parameters that have the types int, long, float, and double. Here are the declarations of the overloaded function Square:

double Square(int i);

double Square(long i);

double Square(float i);

double Square(double i);

Each version of function Square has a different parameter list. When you call the function Square, the compiler examines the data type of the argument to decide which version of function Square to call.

**Example**:

#include<iostream.h>

double Square(int i);

double Square(long i);

double Square(float i);

double Square(double i);

int main()

{

 cout<<Square(1.3)<<endl;

 return 0;

}

double Square(int i)

{ return double(i\*i); }

double Square(long i)

{ return double(i\*i); }

double Square(float i)

{ return double(i\*i); }

double Square(double i)

{ return double(i\*i); }

**Example**:

#include<iostream.h>

int sum(int i, int j);

int sum(int i, int j, int k);

int sum(int i, int j, int k, int l);

int main()

{

 cout<<sum(2,3,5)<<endl;

 return 0;

}

int sum(int i, int j)

{ return i+j; }

int sum(int i, int j, int k)

{ return i+j+k; }

int sum(int i, int j, int k, int l)

{ return i+j+k+l; }

**Exercises**

1. Show the value of x after each of the following statements is performed:

 a) x = fabs( 7.5 )

 b) x = floor( 7.5 )

 c) x = fabs( 0.0 )

 d) x = ceil( 0.0 )

 e) x = fabs( -6.4 )

 f) x = ceil( -6.4 )

 g) x = ceil( -fabs( -8 + floor( -5.5 ) ) )

1. Write a function multiple that determines for a pair of integers whether the second integer is a multiple of the first. The function should take two integer arguments and return true if the second is a multiple of the first, false otherwise. Use this function in a program that inputs a series of pairs of integers.
2. Write a program that inputs a series of integers and passes them one at a time to function even, which uses the modulus operator to determine whether an integer is even. The function should take an integer argument and return true if the integer is even and false otherwise.
3. Write a function that displays at the left margin of the screen a solid square of asterisks whose side is specified in integer parameter side. For example, if side is 4, the function displays

\*\*\*\*

\*\*\*\*

\*\*\*\*

\*\*\*\*

5. Write program segments that accomplish each of the following:

 a) Calculate the integer part of the quotient when integer a is

 divided by integer b.

 b) Calculate the integer remainder when integer a is divided

 by integer b.

 c) Use the program pieces developed in a) and b) to write a

 function that inputs an integer between 1 and 32767 and prints it as a series of digits, each pair of which is separated by two spaces. For example, the integer 4562 should be printed as

4 5 6 2

6. Raising a number n to a power p is the same as multiplying n by itself p times. Write a function called power() that takes a double value for n and an int value for p, and returns the result as a double value. Use a default argument of 2 for p, so that if this argument is omitted, the number n will be squared. Write a main() function that gets values from the user to test this function.

7. Write a function called zeroSmaller() that is passed two int arguments by reference and then sets the smaller of the two numbers to 0. Write a main() program to exercise this function.

8. Write a function that takes two Distance values as arguments and returns the larger one. Include a main() program that accepts two Distance values from the user, compares them, and displays the larger.

9. Write a function called swap() that interchanges two int values passed to it by the calling program. (Note that this function swaps the values of the variables in the calling program, not those in the function.) You’ll need to decide how to pass the arguments. Create a main() program to exercise the function.

10. Write a function that, when you call it, displays a message telling how many times it has been called: “I have been called 3 times”, for instance. Write a main() program that calls this function at least 10 times. Try implementing this function in two different ways. First, use a global variable to store the count. Second, use a local static variable. Which is more appropriate? Why can’t you use a local variable?

11. Write a function that returns the smallest of three double-precision, floating-point numbers.

12. An integer number is said to be a perfect number if the sum of its factors, including 1 (but not the number itself), is equal to the number. For example, 6 is a perfect number, because 6 = 1 + 2 + 3. Write a function **perfect** that determines whether parameter **number** is a perfect number. Use this function in a program that determines and prints all the perfect numbers between 1 and 1000. Print the factors of each perfect number to confirm that the number is indeed perfect. Challenge the power of your computer by testing numbers much larger than 1000.

13. An integer is said to be prime if it is divisible by only 1 and itself. For example, 2, 3, 5 and 7 are primes, but 4, 6, 8 and 9 are not.

 a) Write a function that determines whether a number is

 prime.

 b) Use this function in a program that determines and prints

 all the prime numbers between 1 and 1000.

14. Write a function that takes an integer value and returns the number with its digits reversed. For example, given the number 7631, the function should return 1367.

15. Write a recursive function **power(** **base,** **exponent** **)** that, when invoked, returns

|  |  |
| --- | --- |
|  | base exponent  |
|  | base exponent = base · base exponent -1 |
|  | base1 = base  |

16. The greatest common divisor of integers **x** and **y** is the largest integer that evenly divides both **x** and **y**. Write a recursive function **gcd** that returns the greatest common divisor of **x** and **y**. The **gcd** of **x** and **y** is defined recursively as follows: If **y** is equal to **0**, then **gcd(** **x**, **y** **)** is **x**; otherwise **gcd(** **x,** **y** **)** is **gcd(** **y,** **x** **%** **y** **)**, where **%** is the modulus operator.

**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);

The complete program is as follows:

// arrays as parameters

 #include <iostream>

using namespace std;

 void printarray (int arg[], int length)

 { 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;}

**Pointers**

We have already seen how variables are seen as memory cells that can be accessed using their identifiers. By this way, we did not have to care about the physical location of our data within memory; we simply used its identifier whenever we wanted to refer to our variable. The memory of your computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (one byte). These single-byte memory cells are numbered in a consecutive way, so as, within any block of memory, every cell has the same number as the previous one plus one. This way, each cell can be easily located in the memory because it has a unique address and all the memory cells follow a successive pattern. For example, if we are looking for cell 1776, we know that it is going to be right between cells 1775 and 1777, exactly one thousand cells after 776 and exactly one thousand cells before cell 2776.

**Reference operator (&)**

As soon as, we declare a variable, the amount of memory needed is assigned for it at a specific location in memory (its memory address). We generally do not actively decide the exact location of the variable within the panel of cells that we have imagined the memory to be - Fortunately, that is a task automatically performed by the operating system during runtime. However, in some cases, we may be interested in knowing the address where our variable is being stored during runtime in order to operate with relative positions to it. The address that locates a variable within a memory is what we call a reference to that variable. This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (&), known as reference operator, and which can be literally translated as "address of". For example: x = & y;

This would assign to **x the address of variable** **y**, since when preceding the name of the variable **y** with the reference operator (&), we are no longer talking about the content of the variable itself, but about its reference (i.e., its address in memory).

From now on, we are going to assume that **y** is placed during runtime in the memory address 1776. This number (1776) is just an arbitrary assumption we are inventing right now in order to help clarify some concepts in this tutorial, but in reality, we cannot know before runtime the real value the address of a variable will have in memory.

Consider the following code fragment:

y = 25; d = y; t = &y;

First, we have assigned the value 25 to y (a variable whose address in memory we have assumed to be 1776).

The second statement copied to d the content of variable y (which is 25). This is a standard assignment operation, as we have done so many times before. Finally, the third statement copies to t not to the value contained in y but a reference to it (i.e., its address, which we have assumed to be 1776). The reason is that: in this third assignment operation, we have preceded the identifier y with the reference operator (&), so we were no longer referring to the value of y but to its reference (its address in memory).

The variable that stores the reference to another variable (like t in the previous example) is what we call a **pointer**. **Pointers** are a very powerful feature of the C++ language that has many uses in advanced programming.

Farther ahead, we will see how this type of variable is used and declared.

**Dereference operator (\*)**

We have just seen that a variable which stores a reference to another variable is called a pointer. Pointers are said to "point to" the variable whose reference they store. Using a pointer, we can directly access the value stored in the variable which it points to. To do this, we simply have to precede the pointer's identifier with an asterisk (\*), which acts as **dereference operator** and that can be literally translated to "value pointed by". Therefore, following with the values of the previous example, if we write:

b = \*t; (that we could read as: "b equal to value pointed by t") b would take the value 25, since t is 1776, and the value pointed by 1776 is 25.

**Notice**

The difference between the reference and dereference operators:

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

• \* is the dereference operator and can be read as "value pointed by".

Thus, they have complementary (or opposite) meanings. A variable referenced with & can be dereferenced with \*. Earlier, we performed the following two assignment operations:

y = 25; t = &andy;

Right after these two statements, all of the following expressions would give true as result:

y == 25, &y == 1776, t == 1776, and \*t == 25

The first expression is quite clear considering that the assignment operation performed on y was y=25. The second one uses the reference operator (&), which returns the address of variable y, which we assumed it to have a value of 1776. The third one is somewhat obvious since the second expression was true and the assignment operation performed on ted was t=&y. The fourth expression uses the dereference operator (\*) that, as we have just seen, can be read as "value pointed by", and the value pointed by t is indeed 25. So, after all that, you may also infer that for as long as the address pointed by t remains unchanged the following expression will also be true: \*t == y.

**Declaring variables of pointer types**

Due to the ability of a pointer to directly refer to the value that it points to, it becomes necessary to specify in its declaration which data type a pointer is going to point to. It is not the same thing to point to a char as to point to an int or a float. The declaration of pointers follows this format:

type \* name;

where **type** is the data type of the value that the pointer is intended to point to. This type is not the type of the pointer itself! but the type of the data the pointer points to. For example:

int \* number; char \* character; float \* greatnumber;

These are three declarations of pointers. Each one is intended to point to a different data type, but in fact all of them are pointers and all of them will occupy the same amount of space in memory (the size in memory of a pointer depends on the platform where the code is going to run). Nevertheless, the data to which they point to do not occupy the same amount of space nor are of the same type: the first one points to an int, the second one to a

char and the last one to a float. Therefore, although these three example variables are all of them pointers which occupy the same size in memory, they are said to have different types: int\*, char\* and float\* respectively, depending on the type they point to.

I want to emphasize that the asterisk sign (\*) that we use when declaring a pointer only means that it is a pointer.

**Example**

// my first pointer

#include <iostream>

using namespace std;

 int main ()

{ int firstvalue, secondvalue;

 int \* mypointer;

 mypointer = &firstvalue;

 \*mypointer = 10;

 mypointer = &secondvalue;

 \*mypointer = 20;

 cout << "firstvalue is " << firstvalue << endl;

 cout << "secondvalue is " << secondvalue << endl;

 return 0;}

Notice that even though we have never directly set a value to either firstvalue or secondvalue, both end up with a value set indirectly through the use of mypointer. This is the procedure:

First, we have assigned as value of mypointer a reference to firstvalue using the reference operator (&). And then we have assigned the value 10 to the memory location pointed by mypointer, that because at this moment is pointing to the memory location of firstvalue, this in fact modifies the value of firstvalue.

In order to demonstrate that a pointer may take several different values during the same program I have repeated the process with secondvalue and that same pointer, mypointer.

**Example**

#include <iostream>

using namespace std;

int main ()

 { int firstvalue = 5, secondvalue = 15;

 int \* p1, \* p2;

 p1 = &firstvalue; // p1 = address of firstvalue

 p2 = &secondvalue; // p2 = address of secondvalue

 \*p1 = 10; // value pointed by p1 = 10

 \*p2 = \*p1; // value pointed by p2 = value pointed by p1

 p1 = p2; // p1 = p2 (value of pointer is copied)

 \*p1 = 20; // value pointed by p1 = 20

 cout << "firstvalue is " << firstvalue << endl;

 cout << "secondvalue is " << secondvalue << endl;

 return 0;}

Notice that there are expressions with pointers p1 and p2, both with and without dereference operator (\*). The meaning of an expression using the dereference operator (\*) is very different from one that does not: When this operator precedes the pointer name, the expression refers to the value being pointed, while when a pointer name appears without this operator, it refers to the value of the pointer itself (i.e. the address of what the pointer is pointing to).

Another thing that may call your attention is the line:

int \* p1, \* p2;

This declares the two pointers used in the previous example. But notice that there is an asterisk (\*) for each pointer, in order for both to have type int\* (pointer to int).

Otherwise, the type for the second variable declared in that line would have been int (and not int\*) because of precedence relationships. If we had written:

int \* p1, p2;

p1 would indeed have int\* type, but p2 would have type int (spaces do not matter at all for this purpose). This is due to operator precedence rules. But anyway, simply remembering that you have to put one asterisk per pointer is enough for most pointer users.

**Pointer initialization**

When declaring pointers, we may want to explicitly specify which variable we want them to point to: int number;

int \*tommy = &number;

The behavior of this code is equivalent to:

int number; int \*tommy; tommy = &number;

When a pointer initialization takes place, we are always assigning the reference value to where the pointer points (tommy), never the value being pointed (\*tommy). You must consider that at the moment of declaring a pointer, the

asterisk (\*) indicates only that it is a pointer, it is not the dereference operator (although both use the same sign: \*). Remember, they are two different functions of one sign. Thus, we must take care not to confuse the previous

code with:

int number; int \*tommy; \*tommy = &number;

that is incorrect, and anyway would not have much sense in this case if you think about it.

As in the case of arrays, the compiler allows the special case that we want to initialize the content at which the pointer points with constants at the same moment the pointer is declared:

char \* terry = "hello";

In this case, memory space is reserved to contain "hello" and then a pointer to the first character of this memory block is assigned to terry. If we imagine that "hello" is stored at the memory locations that start at addresses 1702, we can represent the previous declaration as:



It is important to indicate that terry contains the value 1702, and not 'h' nor "hello", although 1702 indeed is the address of both of these.

The pointer terry points to a sequence of characters and can be read as if it was an array (remember that an array is just like a constant pointer). For example, we can access the fifth element of the array with any of these

two expression:

\*(terry+4) or terry[4]

Both expressions have a value of 'o' (the fifth element of the array).

**Pointer arithmetics**

To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer data types. To begin with, only **addition** and **subtraction** operations are allowed to be conducted with them, the others make no sense in the world of pointers. But both addition and subtraction have a different behavior with pointers according to the size of the data type to which they point.

When we saw the different fundamental data types, we saw that some occupy more or less space than others in the memory. For example, let's assume that in a given compiler for a specific machine, char takes 1 byte, short takes 2 bytes and long takes 4.

Suppose that we define three pointers in this compiler:

char \*mychar; short \*myshort; long \*mylong;

and that we know that they point to memory locations 1000, 2000 and 3000 respectively.

So if we write:

mychar++; myshort++; mylong++;

mychar, as you may expect, would contain the value 1001. But not so obviously, myshort would contain the value 2002, and mylong would contain 3004, even though they have each been increased only once. The reason is that when adding one to a pointer we are making it to point to the following element of the same type with which it has

been defined, and therefore the size in bytes of the type pointed is added to the pointer.

****

This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we write:

mychar = mychar + 1; myshort = myshort + 1; mylong = mylong + 1;

Both the increase (++) and decrease (--) operators **have greater operator precedence than the dereference operator (\*),** but both have a special behavior when used as suffix (the expression is evaluated with the value it had before being increased). Therefore, the following expression may lead to confusion:

\*p++

Because ++ has **greater precedence than \***, this expression is equivalent to \*(p++). Therefore, what it does is to increase the value of p (so it now points to the next element), but because ++ is used as postfix, the whole expression is evaluated as the value pointed by the original reference (the address the pointer pointed to before being increased).

Notice the difference with:

(\*p)++

Here, the expression would have been evaluated as the value pointed by p increased by one. The value of p (the pointer itself) would not be modified (what is being modified is what it is being pointed to by this pointer).

If we write:

\*p++ = \*q++;

 Because ++ has a higher precedence than \*, both p and q are increased, but because both increase operators (++) are used as postfix and not prefix, the value assigned to \*p is \*q before

 both p and q are increased. And then both are increased. It would be roughly equivalent to:

\*p = \*q; ++p; ++q;

Like always, I recommend you to use parentheses () in order to avoid unexpected results and to give more legibility to the code.

**Pointers to pointers**

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers). In order to do that, we only need to add an asterisk (\*) for each level of reference in their declarations:

char a;

char \* b;

 char \*\* c;

a = 'z'; b = &a; c = &b;



The new thing in this example is variable c, which can be used in three different levels of indirection, each one of them would correspond to a different value:

• c has type char\*\* and a value of 8092

• \*c has type char\* and a value of 7230

• \*\*c has type char and a value of 'z'

**void pointers**

The void type of pointer is a special type of pointer. In C++, void represents the absence of type, so void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereference properties).

This allows void pointers to point to any data type, from an integer value or a float to a string of characters. But in exchange they have a great limitation: the data pointed by them cannot be directly dereferenced (which is logical, since we have no type to dereference to), and for that reason we will always have to cast the address in the void pointer to some other pointer type that points to a concrete data type before dereferencing it.

One of its uses may be to pass generic parameters to a function:

// increaser #include <iostream>

using namespace std;

 void increase (void\* data, int psize)

{ if ( psize == sizeof(char) )

 { char\* pchar;

 Pchar =(char\*) data; ++(\*pchar); }

 else if (psize == sizeof(int) )

 { int\* pint; pint=(int\*)data; ++(\*pint); }

 }

 int main ()

 { char a = 'x'; int b = 1602;

 increase (&a,sizeof(a));

 increase (&b,sizeof(b));

 cout << a << ", " << b << endl; return 0; }

sizeof is an operator integrated in the C++ language that returns the size in bytes of its parameter. For non dynamic data types this

value is a constant. Therefore, for example, sizeof(char) is 1, because char type is one byte long.

**Null pointer**

A null pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointing to any valid reference or memory address. This value is the result of type-casting the integer value zero to any pointer type.

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

Do not confuse null pointers with void pointers. A null pointer is a **value** that any pointer may take to represent that it is pointing to "nowhere", while a void pointer is a special type of pointer that can point to somewhere without a specific type. One refers to the **value** stored in the pointer itself and the other to the **type** of data it points to.

**Pointers to functions**

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

// pointer to functions

 #include <iostream>

using namespace std;

int addition (int a, int b) { return (a+b); }

 int subtraction (int a, int b) { return (a-b); }

 int operation (int x, int y, int (\*functocall)(int,int))

 { int g; g = (\*functocall)(x,y); return (g); }

 int main ()

 { int m,n;

 int (\*minus)(int,int) = subtraction;

 m = operation (7, 5, addition);

 n = operation (20, m, minus) cout <<n; return 0; }

In the example, minus is a pointer to a function that has two parameters of type int. It is immediately assigned to point to the function subtraction, all in a single line:

int (\* minus)(int,int) = subtraction;

**Pointers and arrays**

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

int numbers [20]; int \* p;

The following assignment operation would be valid:

p = numbers;

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

numbers = p;

Because numbers is an array, so it operates as a constant pointer, and we cannot assign values to constants. Due to the characteristics of variables, all expressions that include pointers in the following example are perfectly valid:

// more pointers

#include <iostream>

using namespace std;

int main ()

 { int numbers[5]; int \* p;

 p = numbers;

\*p = 10; p++;

\*p = 20;

 p = &numbers[2];

\*p = 30;

 p = numbers + 3;

 \*p = 40;

 p = numbers;

\*(p+4) = 50;

 for (int n=0; n<5; n++) cout << numbers[n] << ", "; return 0;}

a[5] = 0; // a [offset of 5] = 0

\*(a+5) = 0; // pointed by (a+5) = 0

These two expressions are equivalent and valid both if a is a pointer or if a is an array.

**Dynamic Memory**

Until now, in all our programs, we have only had as much memory available as we declared for our variables, having the size of all of them to be determined in the source code, before the execution of the program. But, what if we need a variable amount of memory that can only be determined during runtime? For example, in the case that we need some user input to determine the necessary amount of memory space.

The answer is dynamic memory, for which C++ integrates the operators new and delete.

**Operators new and new[]**

In order to request dynamic memory, we use the operator new. new is followed by a data type specifier and -if a sequence of more than one element is required- the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated. Its form is:

pointer = new type = new type [number\_of\_elements]

The first expression is used to allocate memory to contain one single element of type **type**. The second one is used to assign a block (an array) of elements of type **type**, where number\_of\_elements is an integer value representing the amount of these. For example:

int \* bobby; bobby = new int [5];

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



The first element pointed by bobby can be accessed either with the expression bobby[0] or the expression \*bobby. Both are equivalent as has been explained in the section about pointers. The second element can be accessed

either with bobby[1] or \*(bobby+1) and so on...

You could be wondering the difference between declaring a normal array and assigning dynamic memory to a pointer, as we have just done. **The most important difference is that the size of an array has to be a constant value, which limits its size to what we decide at the moment of designing the program, before its execution, whereas the dynamic memory allocation allows us to assign memory during the execution of the program (runtime) using any variable or constant value as its size.**

The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, it is important to have some mechanism to check if our request to allocate memory was successful or not.

C++ provides two standard methods to check if the allocation was successful:

One is by handling exceptions. Using this method an exception of type bad\_alloc is thrown when the allocation fails. Exceptions are a powerful C++ feature explained later in these tutorials. But for now you should know that if

this exception is thrown and it is not handled by a specific handler, the program execution is terminated.

**Operators delete and delete[]**

Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of the operator delete, whose format is:

delete pointer; delete [] pointer;

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements.

The value passed as argument to delete must be either a pointer to a memory block previously allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).

**Example**

// rememb-o-matic

 #include <iostream>

 #include <new>

using namespace std;

 int main ()

{ int i,n; int \* p;

 cout << "How many numbers would you like to type? ";

 cin >> i;

 p= new (nothrow) int[i];

 if (p == 0) cout << "Error: memory could not be allocated"; else { for (n=0; n<i; n++)

 { cout << "Enter number: "; cin >> p[n]; }

 cout << "You have entered: ";

 for (n=0; n<i; n++)

 cout << p[n] << ", "; delete[] p; } return 0;}

**Structures**

A data structure is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different lengths. Data structures are declared in C++ using the following syntax:

struct structure\_name {

 member\_type1 member\_name1;

member\_type2 member\_name2;

member\_type3 member\_name3; . . } object\_names;

where structure\_name is a name for the structure type, object\_name can be a set of valid identifiers for objects that have the type of this structure. Within braces { } there is a list with the data members, each one is specified with a type and a valid identifier as its name. The first thing we have to know is that a data structure creates a new type: Once a data structure is declared, a new type with the identifier specified as structure\_name is created and can be used in the rest of the program as if it was any other type. For example:

struct product { int weight; float price; } product apple;

 product banana, melon;

We have first declared a structure type called product with two members: weight and price, each of a different fundamental type. We have then used this name of the structure type (product) to declare three objects of that type: apple, banana and melon as we would have done with any fundamental data type.

Once declared, product has become a new valid type name like the fundamental ones int, char or short and from that point on we are able to declare objects (variables) of this compound new type, like we have done with apple, banana and melon.

Right at the end of the struct declaration, and before the ending semicolon, we can use the optional field object\_name to directly declare objects of the structure type. For example, we can also declare the structure objects apple, banana and melon at the moment we define the data structure type this way:

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

It is important to clearly differentiate between what is the structure type name, and what is an object (variable) that has this structure type. We can instantiate many objects (i.e. variables, like apple, banana and melon) from a single structure type (product).

Once we have declared our three objects of a determined structure type (apple, banana and melon) we can operate directly with their members. To do that we use a dot (.) inserted between the object name and the member name. For example, we could operate with any of these elements as if they were standard variables of their respective types:

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

Each one of these has the data type corresponding to the member they refer to: apple.weight, banana.weightand melon.weight are of type int, while apple.price, banana.price and melon.price are of type float.

**Pointers to structures**

Like any other type, structures can be pointed by its own type of pointers:

struct movies\_t {string title; int year; }; movies\_t amovie; movies\_t \* pmovie;

Here amovie is an object of structure type movies\_t, and pmovie is a pointer to point to objects of structure type movies\_t. So, the following code would also be valid: pmovie = &amovie;

The value of the pointer pmovie would be assigned to a reference to the object amovie (its memory address).

The arrow operator (->) is a dereference operator that is used exclusively with pointers to objects with members. This operator serves to access a member of an object to which we have a reference.

pmovie->title Which is for all purposes equivalent to (\*pmovie).title

Both expressions pmovie->title and (\*pmovie).title are valid and both mean that we are evaluating the member title of the data structure pointed by a pointer called pmovie. It must be clearly differentiated from \*pmovie.title which is equivalent to \*(pmovie.title).

And that would access the value pointed by a hypothetical pointer member called title of the structure object pmovie (which in this case would not be a pointer). The following panel summarizes possible combinations of pointers and structure members:



**Nesting structures**

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

struct movies\_t { string title; int year; };

struct friends\_t {string name; string email; movies\_t favorite\_movie;} charlie, maria; friends\_t \* pfriends = &charlie;

After the previous declaration we could use any of the following expressions:

charlie.name

maria.favorite\_movie.title

charlie.favorite\_movie.year

pfriends->favorite\_movie.year

**Defined data types (typedef)**

C++ allows the definition of our own types based on other existing data types. We can do this using the keyword typedef, whose format is:

typedef existing\_type new\_type\_name ;

where existing\_type is a C++ fundamental or compound type and new\_type\_name is the name for the new type we are defining. For example:

typedef char C; typedef unsigned int WORD; typedef char \* pChar; typedef char field [50];

In this case we have defined four data types: C, WORD, pChar and field as char, unsigned int, char\* and char[50] respectively, that we could perfectly use in declarations later as any other valid type:

C mychar, anotherchar, \*ptc1;

WORD myword;

pChar ptc2;

field name;

typedef does not create different types. It only creates synonyms of existing types. That means that the type of myword can be considered to be either WORD or unsigned int, since both are in fact the same type.

typedef can be useful to define an alias for a type that is frequently used within a program. It is also useful to define types when it is possible that we will need to change the type in later versions of our program, or if a type you want to use has a name that is too long or confusing.

**Object Oriented Programming**

A class is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword class, with the following format:

class class\_name {

 access\_specifier\_1: member1;

 access\_specifier\_2: member2; ... } object\_names;

Where class\_name is a valid identifier for the class, object\_names is an optional list of names for objects of this class. The body of the declaration can contain members, that can be either data or function declarations, and optionally access specifiers. All is very similar to the declaration on data structures, except that we can now include also functions and members, but also this new thing called access specifier. An access specifier is one of the following three keywords: private, public or protected. These specifiers modify the access rights that the members following

them acquire:

• private members of a class are accessible only from within other members of the same class or from their friends.

• protected members are accessible from members of their same class and from their friends, but also from members of their derived classes.

• Finally, public members are accessible from anywhere where the object is visible. By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before one other class specifier automatically has private access. For example:

class CRectangle {

 int x, y; public:

 void set\_values (int,int); int area (void); } rect;

Declares a class (i.e., a type) called CRectangle and an object (i.e., a variable) of this class called rect. This class contains four members: two data members of type int (member x and member y) with private access (because private is the default access level) and two member functions with public access: set\_values() and

area(), of which for now we have only included their declaration, not their definition. Notice the difference between the class name and the object name: In the previous example, CRectangle was the class name (i.e., the type), whereas rect was an object of type CRectangle. It is the same relationship int and a have in the following declaration:

int a;

where int is the type name (the class) and a is the variable name (the object). After the previous declarations of CRectangle and rect, we can refer within the body of the program to any of the public members of the object rect as if they were normal functions or normal variables, just by putting the object's name followed by a dot (.) and then the name of the member. All very similar to what we did with plain data structures before. For example: rect.set\_values (3,4); myarea = rect.area();

The only members of rect that we cannot access from the body of our program outside the class are x and y, since they have private access and they can only be referred from within other members of that same class. Here is the complete example of class CRectangle:

#include <iostream.h>

 class CRectangle { int x, y; public: void set\_values (int,int); int area () {return (x\*y);} };

void CRectangle::set\_values (int a, int b) { x = a; y = b; }

 int main () { CRectangle rect; rect.set\_values (3,4); cout << "area: " << rect.area(); return 0; }

 The most important new thing in this code is the operator of scope (::, two colons) included in the definition of set\_values(). It is used to define a member of a class from outside the class definition itself.

You may notice that the definition of the member function area() has been included directly within the definition of the CRectangle class given its extreme simplicity, whereas set\_values() has only its prototype declared within the class, but its definition is outside it. In this outside declaration, we must use the operator of scope (::) to specify that we are defining a function that is a member of the class CRectangle and not a regular global function.

The scope operator (::) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, in the function set\_values() of the previous code, we have been able to use the variables x and y, which are private members of class CRectangle, which means they are only accessible from other members of their class. The only difference between defining a class member function completely within its class or to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second it will be a normal (not-inline) class member function, which in fact supposes no difference in behavior. Members x and y have private access (remember that if nothing else is said, all members of a class defined with keyword class have private access). By declaring them private we deny access to them from anywhere outside the class. This makes sense, since we have already defined a member function to set values for those members within the object: the member function set\_values(). Therefore, the rest of the program does not need to have direct

access to them. Perhaps in a so simple example as this, it is difficult to see an utility in protecting those two variables, but in greater projects it may be very important that values cannot be modified in an unexpected way (unexpected from the point of view of the object). One of the greater advantages of a class is that, as any other type, we can declare several objects of it.

The only difference between the structure and class is that: members of classes declared with the keyword struct have public access by default, while members of classes declared with the keyword class have private access. For all other purposes both keywords are equivalent.

**Constructors and Destructors**

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the member function area() before having called function set\_values()? Probably we would have gotten an undetermined result since the members x and y would have never been assigned a value. In order to avoid that, a class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and cannot have any return type; not even void. We are going to implement CRectangle including a constructor:

#include <iostream> using namespace std;

 class CRectangle { int width, height; public: CRectangle (int,int); int area () {return (width\*height);} }; CRectangle::CRectangle (int a, int b) { width = a; height = b; } int main () { CRectangle rect (3,4); CRectangle rectb (5,6); cout << "rect area: " << rect.area() << endl; cout << "rectb area: " << rectb.area() << endl; return 0; }

As you can see, the result of this example is identical to the previous one. But now we have removed the member function set\_values(), and have included instead a constructor that performs a similar action: it initializes the values of x and y with the parameters that are passed to it. Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created. Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created. You can also see how neither the constructor prototype declaration (within the class) nor the latter constructor definition includes a return value; not even void. The destructor fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator delete. The destructor must have the same name as the class, but preceded with a tilde sign (~) and it must also return no value. The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

**Overloading Constructors**

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. Remember that for overloaded functions the compiler will call the one whose parameters match the arguments used in the function call. In the case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on the object declaration:

**Default constructor**

If you do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments. Therefore, after declaring a class like this one:

class CExample { public: int a,b,c; void multiply (int n, int m) { a=n; b=m; c=a\*b; }; };

The compiler assumes that CExample has a default constructor, so you can declare objects of this class by simply declaring them without any arguments: CExample ex;

 But as soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. So you have to declare all objects of that class according to the constructor prototypes you defined for the class:

**Pointers to classes**

It is perfectly valid to create pointers that point to classes. We simply have to consider that once declared, a class becomes a valid type, so we can use the class name as the type for the pointer. For example:

CRectangle \* prect;

is a pointer to an object of class CRectangle.

As it happened with data structures, in order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (->) of indirection.