



Programming with C++ Language

**Lectures for First Class
Computer Science Department**

By

Israa A. Alwan

2017

Flowcharts and Algorithms

Algorithm

Sequence of step-by-step instructions that will produce a solution to a problem.

Examples:

The taxi algorithm:

- Go to the taxi stand.
- Get in a taxi.
- Give the driver my address.

The call-me algorithm:

- When your plane arrives, call my cell phone.
- Meet me outside baggage claim.

Q. Define the algorithm for mailing a letter, from obtaining the envelope to placing it in the mailbox

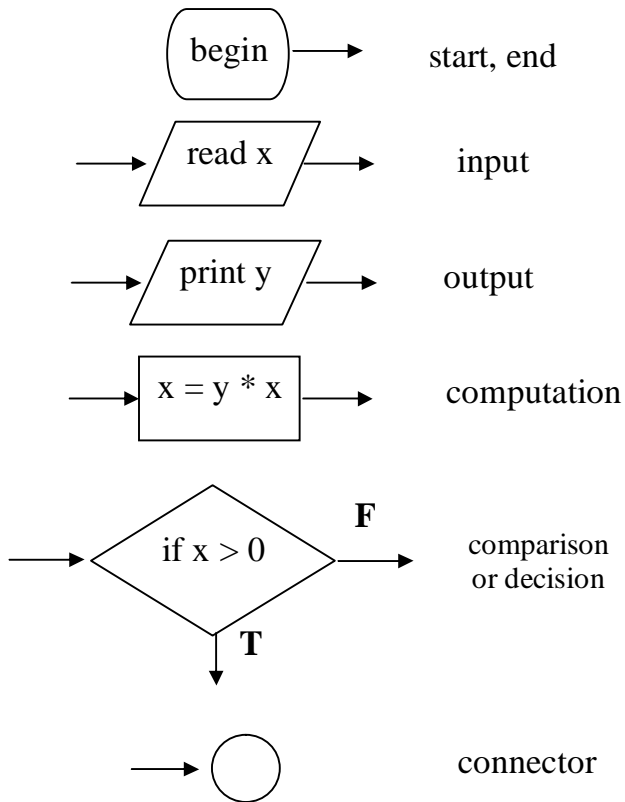
- Obtain Envelope
- If letter is too big, fold letter
- Place letter in envelope
- Address Envelope
- Seal envelope
- Obtain stamp
- Affix stamp
- Place in mailbox

Flowcharts

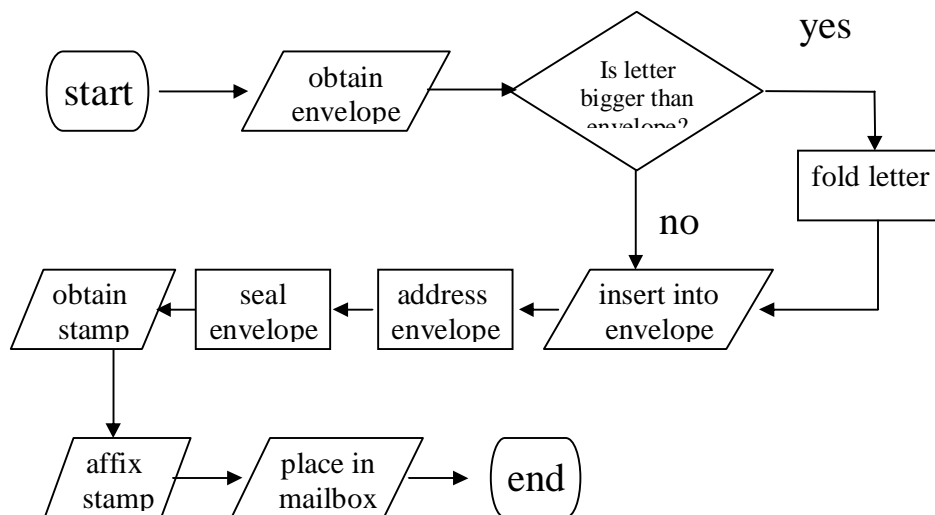
Flowcharts are graphical and verbal illustrations of algorithms.

Flowchart Elements

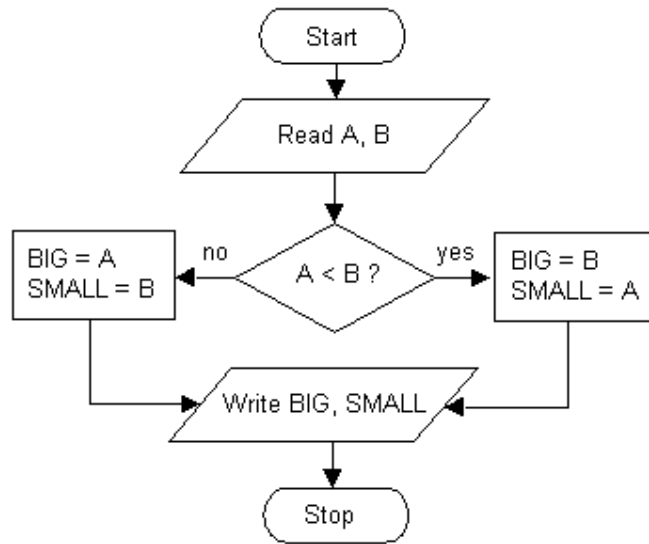
Specific shapes have specific roles in a flowchart:



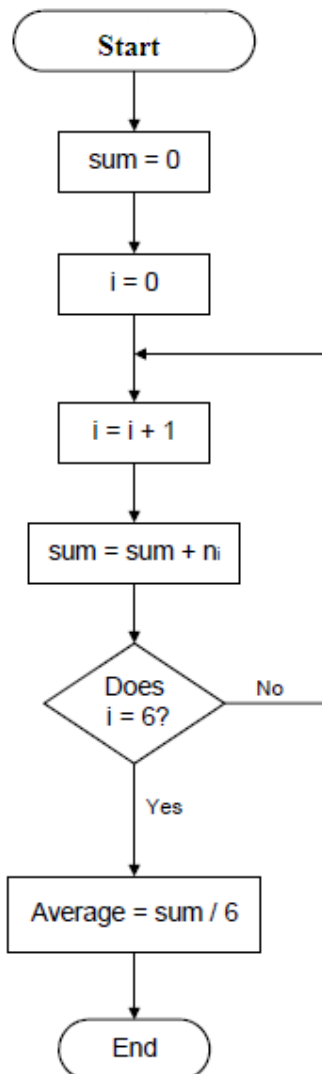
Q. Draw the flowchart for mailing a letter



Q. Draw the flowchart for reading two numbers and displaying the numbers in decreasing order:



Q. Draw the flowchart for finding the average of your six lessons





Definition

C++ is an improved version of C that takes the C language to the next level of evolution of programming languages—those that provide *object-oriented programming*.

C++ Compared with Other Languages

C++ is a *structured language* that allows large programs to be built out of small, easy to understand pieces of code. Early languages, such as the original BASIC and FORTRAN, did not have this idea. To write large programs was difficult and the results of trying are described as *spaghetti code*. Many of the object-oriented features of C++ have been introduced to address this problem. C++ has many of the features of a high-level language (a programming language that uses commands that bear little relationship to the instructions a computer uses), but it also can handle the same programming detail as assembler language (code that directly represents machine instructions, which is a low-level language).

How to write a simple program with C++?

The main Structure of C++ Program is as bellow

```
#include "iostream.h"
void main()
{
    Declaration
    C++ statements
}

func1
func2
```





#include "iostream.h"

#include is a directive, which tells the compiler that we want to use the **iostream** library. The name inside quotes is a header. Every program that uses a library facility must include its associated header. The **#include** directive must be written on a single line the name of the header and the **#include** must appear on the same line. In general, **#include** directives should appear outside any function. Typically, all the **#include** directives for a program appear at the beginning of the file. The **iostream** library defines versions of the input and output operators that accept all of the built-in types.

void main()

A **main()** is a function consists of a sequence of statements that perform the work of the function. The operating system executes a program by calling the function named **main**. That function executes its constituent statements and returns a value to the operating system.

{  Marks the **beginning** of the body of the function
}  Indicates the **end** of the body of the function



All users' defined functions must be written after the end of the program.



Semicolons mark the end of most statements in C++. They are easy to overlook, but when forgotten can lead to mysterious compiler error messages.

Exercise



What is the origin of C++ Language?



Primitive Built-in Types

C++ defines a set of **arithmetic types**, which represent **integers**, **floating-point** numbers, and individual **characters** and **boolean** values. The size of the arithmetic types **varies** across **machines**. By size, we mean the number of bits used to represent the type. The standard guarantees a minimum size for each of the arithmetic types, but it does not prevent compilers from using larger sizes. Indeed, almost all compilers use a larger size for **int** than is strictly required.

Type	Meaning	Minimum Size
bool	boolean	NA
char	character	1 Byte (8 Bits)
short	short integer	2 Byte (16 bits)
int	integer	According to the system
long	long integer	4 Byte (32 bits)
float	single-precision floating-point	4 Byte (32 Bits)
double	double-precision floating-point	8 Byte (64 Bits)
long double	extended-precision floating-point	10 Byte (80 Bits)



Because the number of bits varies, the maximum (or minimum) values that these types can represent also vary by machine.



By default, an **integer variable** is assumed to be signed (i.e., have a signed representation so that it can assume positive as well as negative values). However, an integer can be defined to be unsigned by using the keyword **unsigned** in its definition. The keyword **signed** is also allowed but is redundant.

```
unsigned short age = 20;
unsigned int salary = 65000;
unsigned long price = 4500000;
```



The **float** type is usually not precise enough for real programs float is guaranteed to offer only 6 significant digits. The **double** type guarantees at least 10 significant digits, which is sufficient for most calculations.



Arithmetic Operators

C++ provides five basic arithmetic operators.

Operator	Name	Example
+	Addition	12 + 4.9 // gives 16.9
-	Subtraction	3.98 - 4 // gives -0.02
*	Multiplication	2 * 3.4 // gives 6.8
/	Division	9 / 2.0 // gives 4.5
%	Remainder	13 % 3 // gives 1



Except for remainder (%) all other arithmetic operators can accept a mix of integer and real operands. Generally, if both operands are integers then the result will be an integer. However, if one or both of the operands are reals then the result will be a real (or **double** to be exact).



When both operands of the division operator (/) are integers then the division is performed as an **integer division** and not the normal division we are used to. Integer division always results in an integer outcome (i.e., the result is always rounded down). For example:

```
9 / 2 // gives 4, not 4.5!
-9 / 2 // gives -5, not -4!
```



Unintended integer divisions are a common source of programming errors. To obtain a real division when both operands are integers, you should cast one of the operands to be real:

```
int cost = 100;
int volume = 80;
double unitPrice = cost / (double) volume; // gives 1.25
```



In addition to the above arithmetic operators, there are the **unary minus** and the **Unary plus** operators. the unary minus operator negates its operand:

```
int i = 1024;
int k = -i; // negates the value of its operand
```

Unary plus returns the operand itself. It makes no change to its operand.



It is illegal to divide a number by zero. This results in a run-time division-by-zero failure which typically causes the program to terminate.



Relational Operators

C++ provides **six** relational operators for comparing numeric quantities. These are shown below. Relational operators evaluate to **1** (representing the **true** outcome) or **0** (representing the **false** outcome).

Operator	Name	Example
==	Equality	5 == 5 // gives 1
!=	Inequality	5 != 5 // gives 0
<	Less Than	5 < 5.5 // gives 1
<=	Less Than or Equal	5 <= 5 // gives 1
>	Greater Than	5 > 5.5 // gives 0
>=	Greater Than or Equal	6.3 >= 5 // gives 1



The <= and >= operators are only supported in the form shown. In particular, =< and => are both invalid and do not mean anything.



The operands of a relational operator must evaluate to a number. Characters are valid operands since they are represented by numeric values. For example (assuming ASCII coding):

'A' < 'F' // gives 1 (is like 65 < 70)



The relational operators should not be used for comparing **strings**, because this will result in the string addresses being compared, not the string contents. For example, the expression

"HELLO" < "BYE"

causes the address of "HELLO" to be compared to the address of "BYE". As these addresses are determined by the compiler (in a machine-dependent manner), the outcome may be 0 or may be 1, and is therefore undefined. C++ provides library functions (e.g., **strcmp**) for the comparison of string. These will be described later.



Logical Operators

C++ provides **three** logical operators for combining logical expression. Like the relational operators, logical operators evaluate to **1** or **0**.

Operator	Name	Example
!	Logical Negation	!(5 == 5) // gives 0
&&	Logical And	5 < 6 && 6 < 6 // gives 1
	Logical Or	5 < 6 6 < 5 // gives 1



Logical negation is a unary operator, which negates the logical value of its single operand. If its operand is nonzero it produce 0, and if it is 0 it produces 1.



Logical and produces 0 if one or both of its operands evaluate to 0. Otherwise, it produces 1. Logical or produces 0 if both of its operands evaluate to 0. Otherwise, it produces 1.



Here we talk of zero and nonzero operands (not zero and 1). In general, any nonzero value can be used to represent the logical true, whereas only zero represents the logical false. The following are, therefore, all valid logical expressions:


```
!20 // gives 0  
10 && 5 // gives 1  
10 || 5.5 // gives 1  
10 && 0 // gives 0
```



Increment/Decrement Operators


The auto increment (++) and auto decrement (--) operators provide a convenient way of, respectively, adding and subtracting **1** from a numeric variable. These are shown below. The examples assume the following variable definition: **int k = 5;**


Operator	Name	Example
++	Auto Increment (prefix)	++k + 10 // gives 16
++	Auto Increment (postfix)	k++ + 10 // gives 15
--	Auto Decrement (prefix)	--k + 10 // gives 14
--	Auto Decrement (postfix)	k-- + 10 // gives 15


 ++k; or k++; \longrightarrow k=k+1;
--k; or k--; \longrightarrow k=k-1;

A=5;
X=A++; \longrightarrow X=5, A=6

A=5;
X=++A; \longrightarrow X=6, A=6

 Both operators can be used in **prefix** and **postfix** form. The difference is significant. When used in prefix form, the operator is first applied and the outcome is then used in the expression. When used in the postfix form, the expression is evaluated first and then the operator applied.

 Both operators may be applied to integer as well as real variables, although in practice real variables are rarely useful in this form.

 Use **Postfix** Operators Only When Necessary. The prefix version does less work (simple). It increments the value and returns the incremented version. The postfix operator must store the original value so that it can return the unincremented value as its result. For **ints** and pointers, the compiler can optimize away this extra work. For more complex iterator types, this extra work potentially could be more costly.

Exercise



Why do you think C++ wasn't named ++C?

Exercise



Explain the difference between prefix and postfix increment.



Assignment Operator

The assignment operator is used for storing a value at some memory location (typically denoted by a variable). Its left operand should be an lvalue, and its right operand may be an arbitrary expression. The latter is evaluated and the outcome is stored in the location denoted by the lvalue. An **lvalue** (standing for **left value**) is anything that denotes a memory location in which a value may be stored.

Operator	Example	Equivalent to
=	n = 25	
+=	n += 25	n = n + 25
-=	n -= 25	n = n - 25
*=	n *= 25	n = n * 25
/=	n /= 25	n = n / 25
%=	n %= 25	n = n % 25



An assignment operation is itself an expression whose value is the value stored in its left operand. An assignment operation can therefore be used as the right operand of another assignment operation. **Any number of assignments** can be concatenated in this fashion to form one expression. For example:

```
int m, n, p;
m = n = p = 100;      // means: n = (m = (p = 100));
m = (n = p = 100) + 2; // means: m = (n = (p = 100)) + 2;
```

This is equally applicable to other forms of assignment. For example:

```
m = 100;
m += n = p = 10;      // means: m = m + (n = p = 10);
```



Beware of Confusing **Equality** and **Assignment** Operators. The fact that we can use assignment in a condition can have surprising effects:

```
if (i = 42)
```

This code is legal: What happens is that 42 is assigned to *i* and then the result of the assignment is tested. In this case, 42 is nonzero, which is interpreted as a true value. The author of this code almost surely intended to test whether *i* was 42:

```
if (i == 42)
```

Bugs of this sort are notoriously difficult to find. Some, but not all, compilers are kind enough to warn about code such as this example.

Exercise



Explain what happens in each of the `if` tests:

```
if (42 = i) // ...
if (i = 42) // ...
```



Operator Precedence

The order in which operators are evaluated in an expression is significant and is determined by **precedence rules**. These rules divide the C++ operators into a number of precedence levels as shown below. Operators in higher levels take precedence **over** operators in lower levels.

Level	Operator	Kind	Order of Evaluation
Highest	::	Unary	Both
	() [] .	Binary	Left to Right
	+ ++ ! * - -- &	Unary	Right to Left
	* / %	Binary	Left to Right
	+ -	Binary	Left to Right
	< <= > >=	Binary	Left to Right
	== !=	Binary	Left to Right
	&&	Binary	Left to Right
		Binary	Left to Right
	= += *= %= - = /=	Binary	Right to Left
Lowest	,	Binary	Left to Right

For example, in

$$a == b + c * d$$

$c * d$ is evaluated first because $*$ has a higher precedence than $+$ and $==$. The result is then added to b because $+$ has a higher precedence than $==$, and then $==$ is evaluated.



Precedence rules can be overridden using brackets or Parentheses. For example, rewriting the above expression as

$$a == (b + c) * d$$

causes $+$ to be evaluated before $*$.



Operators with the same precedence level are evaluated in the order specified by the last column of the above Table. For example, in

$$a = b += c$$

the evaluation order is right to left, so first $b += c$ is evaluated, followed by $a = b$



Misunderstanding how expressions and operands are evaluated is a rich source of bugs. Moreover, the resulting bugs are difficult to find because reading the program does not reveal the error unless the programmer already understands the rules. **So, when in doubt, parenthesize expressions to force the grouping that the logic of your program requires.**

Exercise



Parenthesize the following expression to indicate how it is evaluated. Test your answer by compiling the expression and printing its result.

$$12 / 3 * 4 + 5 * 15 + 24 \% 4 / 2$$

Exercise



Write expressions for the following:

- **To test if a number n is even.**
- **To test if a character c is a digit.**
- **To test if a character c is a letter.**
- **To do the test: n is odd and positive or n is even and negative.**
- **To give the absolute value of a number n .**

Comments

Program text that is ignored by the compiler. C++ has two kinds of comments: single-line and paired. Single-line comments start with a `//`. Everything from the `//` to the end of the line is a comment. Paired comments begin with a `/*` and include all text up to the next `*/`.

Exercise



Which form is better?



Variables

A variable is a symbolic name for a memory location in which data can be stored and subsequently recalled. Variables are used for holding data values so that they can be utilized in various computations in a program. All variables have two important attributes:

- A **type** which is established when the variable is defined (e.g., integer, float, character), where determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable. Once defined, the type of a C++ variable cannot be changed.
- A **value** which can be changed by assigning a new value to the variable. The kind of values a variable can assume depends on its type. For example, an integer variable can only take integer values (e.g., 2, 100, -12).



As a general rule, a variable is defined by specifying its **type first**, followed by the **variable name**, followed by a **semicolon**. For example:-

```
#include "iostream.h"
void main ()
{
    int workDays;
    float workHours, payRate, weeklyPay;
    // do something
}
```

As illustrated by the above example, multiple variables of the same type can be defined at once by separating them with **commas**.



It is possible to define a variable and initialize it at the same time. This is considered a good programming practice, because it pre-empts the possibility of using the variable prior to it being initialized. For example:-

```
#include "iostream.h"
void main ()
{
    int workDays = 5;
    float workHours = 7.5;
    float payRate = 38.55;
    // do something
}
```



The name of a variable can be composed of **letters**, **digits**, and the **underscore** character. It must begin with either a letter or an underscore. Upper- and lowercase letters are distinct: Identifiers in C++ are case-sensitive. The following defines four distinct identifiers:

```
// declares four different int variables  
int somename, someName, SomeName, SOMENAME;  
int x,X;
```

Exercise



Which, if any, of the following names are invalid? Correct each identified invalid name.

- (a) `int double = 3.14159;`
- (b) `char _;`
- (c) `bool catch-22;`
- (d) `char 1_or_2 ='1';`
- (e) `float Float = 3.14f;`



Type Casting

Converting an expression of a given type into another type is known as **type-casting**.

We have already seen some ways to type cast:

Implicit conversion

Implicit conversions **do not require any operator**. They are **automatically performed** when variables of one type are **mixed** with variables of another type. For example:



//Implicit type conversion example

```
#include "iostream.h"
void main()
{
    int x=2;
    int i=10.5; // i receives 10
    float f=3.14;
    double d;
    char ch;
    long l;
    unsigned short s;
    d=x; // converts 2 to a double to give 2.0
    x=f; // converts 3.14 to an int to give 3
    l=f; // converts 3.14 to a long to give 3L
    ch=65; // converts 65 to a char whose code is 65 ('A')
    s=f; // gives 3 as an unsigned short
    cout<<" The values of variables after implicit conversion\n";
    cout<<" i = "<<i<<"\n";
    cout<<" d = "<<d<<"\n";
    cout<<" x = "<<x<<"\n";
    cout<<" l = "<<l<<"\n";
    cout<<" ch = "<<ch<<"\n";
    cout<<"s = "<<s<<"\n";
    cout<<"more implicit type conversion\n";
    ch+=1; //converts 65+1 to a char whose code is 66 ('B')
    cout<<" ch after increment = "<<ch<<"\n";
    d=1; // d receives 1.0
    cout<<" d="<<d<<"\n";
    i = i + d; // means: i = int(double(i) + d)
    cout<<" i=i+d is"<<i<<"\n";
}
```



Arithmetic operations are applicable with variables that have a character data type.



Some of **implicit conversions** may imply a loss of precision, which the compiler can signal with a warning. This can be avoided with an **explicit conversion**.

Explicit conversion

A value in any of the built-in types, can be converted (type-cast) to any of the other types. For example:

```
(int) 3.14 // converts 3.14 to an int to give 3
(long) 3.14 // converts 3.14 to a long to give 3L
(double) 2 // converts 2 to a double to give 2.0
(char) 122 // converts 122 to a char whose code is 122
(unsigned short) 3.14 // gives 3 as an unsigned short
```

As shown by these examples, the built-in type identifiers can be used as **type operators**. Type operators are **unary** (i.e., take one operand) and **appear inside brackets to the left of their operand**. This is called **explicit type conversion**. When the type name is just **one word**, an **alternate notation** may be used in which the **brackets appear around the operand**:

```
int(3.14) // same as: (int) 3.14
↑
↑
// c-like cast notation
// functional notation
```



C++ has **two notations** for explicit type conversion: functional and c-like casting.

Exercise



What will be the value of each of the following variables after its initialization?

```
float p = 3.14; int x = p;
double d = 2 * x;
long k = p - 3;
char c = 'a' + 2;
char c = 'p' + 'A' - 'a';
```



Basic Input/Output

Using the standard input and output library, we will be able to interact with the user by printing messages on the screen and getting the user's input from the keyboard. The standard C++ library includes the header file **iostream**, where the standard input and output stream objects are declared.

Standard Output (cout)

By default, the standard output of a program is the **screen**, and the C++ stream object defined to access it is **cout**.

cout is used in conjunction with the **insertion operator**, which is written as `<<`. For example:-

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



The sentence in the first instruction is enclosed between double quotes (") because it is a constant string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes (") so that they can be clearly distinguished from variable names. For example, these two sentences have very different results:

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



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

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

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

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

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

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



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

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

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

This is a sentence. This is another sentence.

In order to perform a line break on the output we must explicitly insert a new-line character into **cout**. In C++ a new-line character can be specified as `\n`:

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

This produces the following output:

First sentence.
Second sentence.
Third sentence.



There are more characters can be used with **cout**:

- `\t` Tab character
- `\b` Backspace character
- `\r` Forces the cursor to move or return to the beginning of the current line.

For example:-

```
cout<<"\nJoe\tFred\tSally\r";
```

This produces the following output:

New line
Joe Fred sally

Exercise



What is the output of the following statements:

- 1- `cout<<"\nHere, something is wrong\r"<<" "`;
- 2- `cout<<"If you fine, I will be fine\b"<<" "<<"\n";`



Standard Input (cin)

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

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

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



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



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

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

is equivalent to:

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

In both cases the user must give two data, one for variable **a** and another one for variable **b** that may be separated by any valid blank separator: a space, a tab character or a newline.



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



Insertion operator (`<<`) for **cout**.
Extraction operator (`>>`) for **cin**.



I/O Example

```
#include "iostream"  
void main ()  
{  
    int i;  
    cout << "Please enter an integer value: ";  
    cin >> i;  
    cout << "The value you entered is " << i;  
}
```



Write a C++ program to read an integer number and print its original and double values.



Control Structures

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, **repeat** code or **take decisions**. For that purpose, C++ provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

Conditional structure: if and else

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

```
if (condition)
    statement;
```

Where condition is the expression that is being evaluated. If this condition is **true**, statement is **executed**. If it is **false**, statement is **ignored** (not executed) and the program continues right after this conditional structure. For example, the following code fragment prints **x is 100** only if the value stored in the **x** variable is indeed **100**:

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

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

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



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

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

For example:

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

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



Statement2 can be any statement or a **block of statements** enclosed in curly braces { }.

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

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



C++ compiler handles the **Nested if** by matching **else** to the **nearest if**, not according to the semicolon that found at the end of the statement that precedes the **else**, as in Pascal language.



Common Mistakes of if statement

There are several common mistakes in the writing of **if** statements (by the users). Some of these mistakes may result in compiler errors and therefore are easy to spot. However, other mistakes are harder to pick out since they do not cause an error, either at compile time or run-time, but instead give rise to illogical results. These mistakes are:



Don't Put a Semicolon after the Relational Expression!

```
if ( num % 2 == 0 ); // don't put a semicolon here!  
  
    cout << "The number is even\n";
```

Since the compiler generally ignores blank spaces, the following if statement would be the same, and better illustrates visually the problem:

```
if ( num % 2 == 0 )  
    ; // don't put a semicolon here!  
    cout << "The number is even\n";
```

No compiler error will result. The compiler will assume from the semicolon that it is an empty statement. An empty statement does nothing, and though it is perfectly legal in C++, and indeed sometimes has a purpose, here it is not intended.

One consequence will be that the empty statement will execute if the relational expression is true. If this comes about, nothing will happen. So far, there is no harm done.

However, there is an additional consequence, an illogical result. The **cout** statement "The number is even" will execute whether or not the relational expression is true. In other words, even if an odd number is entered, the program will output "The number is even."



Curly Braces Needed for Multiple Conditional Statements. Unless you use curly braces, only the first statement following the if keyword and relational expression is conditional. For example, in the following code, only the first cout statement is conditional. The second cout statement is not, so it will execute whether the relational expression is true or false:

```
if ( num % 2 == 0 )
    cout << "The number is even\n";
    cout << "And the number is not odd\n";
```

Thus, if the user enters an odd number such as 17, the cout statement “The number is even” will not display because the relational expression is false. However, the following statement “And the number is not odd” will display because that statement does not belong to the if statement.

Enter a whole number: 17
And the number is not odd

If you want more than one statement to be part of the overall if statement, you must encase these statements in curly braces:

```
if ( num % 2 == 0 )
{
    cout << "The number is even\n";
    cout << "And the number is not odd\n";
}
```

Now the second cout statement will execute only if the if expression is true.

Forgetting these curly braces when you want multiple statements to be conditional is another common syntax error.



Don't Mistakenly Use the Assignment Operator! The third most common syntax error is to use the assignment operator instead of the relational equality operator because the assignment operator looks like an equal sign:

```
if ( num % 2 = 0 ) // wrong operator!
    cout << "The number is even\n";
```

The result is that the if expression will not evaluate as the result of a comparison. Instead, it will evaluate the expression within the parentheses as the end result of the assignment, with a non-zero value being regarded as true, a zero value being regarded as false.



Common Mistakes of else statement

Just as with the **if** statement, there are several common syntax mistakes with the **else** statement:



No else without an if. You can have an if expression without an else part. However, you cannot have an else part without an if part. The else part must be part of an overall if statement. This requirement is logical. The else part works as “none of the above”; without an if part there is no “above.”



Don't Put a Relational Expression after the else Keyword! Another common mistake is to place a relational expression in parentheses after the else keyword. This will not cause a compiler or run-time error, but it will often cause an illogical result.

```
if ( num % 2 == 0 )
    cout << "The number is even\n";
else ( num % 2 == 1 )
    cout << "The number is odd\n";
```

The program will not compile, and the cout statement following the else expression will be highlighted with an error description such as “missing ‘;’ before identifier ‘cout’.”

Actually, the error description is misleading. There is nothing wrong with the cout statement. Instead, no relational expression should follow the else keyword. The reason is that the else acts like “none of the above” in a multiple choice test. If the if expression is not true, then the conditional statements connected to the else part execute.



Don't Put a Semicolon after the Else! Another common mistake is to place a semicolon after the else expression. This too will not cause a compiler or run-time error, but often will cause an illogical result. For example, in the following code, the cout statement “The number is odd” will output even if the number that's input is even.

```
if ( num % 2 == 0 )
    cout << "The number is even\n";
else; // don't put a semicolon here!
    cout << "The number is odd\n";
```

The result of inputting an even number will be

```
Enter a whole number: 16
The number is even
The number is odd
```



The cout statement “The number is odd” will execute whether or not the relational expression is true because the cout statement no longer is part of the if statement.



Curly Braces Are Needed for Multiple Conditional Statements. As with the if expression, if you want more than one conditional statement to belong to the else part, then you must encase the statements in curly braces. For example, in the following code fragment, the cout statement “This also belongs to the else part” will always display whether the number is even or odd since it does not belong to the if statement.

```
if ( num % 2 == 0 )
    cout << "The number is even\n";
else
    cout << "The number is odd\n";
    cout << "This also belongs to the else part";
```

The sample input and output could be

```
Enter a whole number: 16
The number is even
This also belongs to the else part
```

Encasing the multiple conditional statements in curly braces solves this issue.

```
if ( num % 2 == 0 )
    cout << "The number is even\n";
else
{
    cout << "The number is odd\n";
    cout << "This also belongs to the else part";
}
```



As mentioned before, C++ compiler handles the Nested **if** by matching **else** to the nearest **if**. In the example below:

```
if (minVal <= ivec[i])
    if (minVal == ivec[i])
        ++occurs;
else {
    minVal = ivec[i];
    occurs = 1;
}
```

else part matches the inner if, but in our program we intend else part matches the outer if, What will we do?. This case or problem is called **Dangling else**.



We can force an **else** to match an **outer if** by enclosing the **inner if** in a compound statement:

```

if (minVal <= ivec[i])
{
    if (minVal == ivec[i])
        ++occurs;
}
else {
    minVal = ivec[i];
    occurs = 1;
}

```

Exercise



Correct each of the following:

- (a) `if (ival1 != ival2)`
`ival1 = ival2`
`else ival1 = ival2 = 0;`
- (b) `if (ival < minval)`
`minval = ival; // remember new minimum`
`occurs = 1; // reset occurrence counter`
- (c) `if (ival = 0)`
`ival = x;`

Exercise



What is a "dangling else"? How are else clauses resolved in C++?

Exercise



Assuming that `n` is 20, what will the following code fragment output when executed?

```

if (n >= 0)
    if (n < 10)
        cout << "n is small\n";
else
    cout << "n is negative\n";

```



Write a program which read two chars and if they are 'm', 'r' output "Good Morning", if 'e', 'v' output "Good Evening", otherwise output "Goodbye".



The switch Statement

The switch statement provides a way of choosing between a set of alternatives, based on the value of an expression. The general form of the switch statement is:

```
switch (expression)
{
    case constant1:
        group of statements 1; // zero, one, or more statements
        break;
    case constant2:
        group of statements 2; // zero, one, or more statements
        break;
    .
    .
    .
    default:
        default group of statements; // zero, one, or more statements
}
```

Why switch Statement?

Deeply nested **if else** statements can often be correct syntactically and yet not correctly reflect the programmer's logic. For example, mistaken **else if** matchings are more likely to **pass unnoticed**. Adding a new condition and associated logic or making other changes to the statements is also hard to get right. A **switch statement provides a more convenient way to write deeply nested if/else logic**.

How it works?

It works in the following way: switch evaluates expression and checks if it is equivalent to **constant1**, if it is, it executes group of **statements 1** until it finds the **break** statement. When it finds this **break** statement the program jumps to the end of the switch statement.

If expression was not equal to **constant1** it will be checked against **constant2**. If it is equal to this, it will execute group of **statements 2** until a **break** keyword is found, and then will jump to the end of the switch statement.

Finally, if the value of expression did not match any of the previously specified constants (you can include as many case labels as values you want to check), the program will execute the **statements** included after the **default: label**, if it exists (**since it is optional**).



Both of the following code fragments have the same behavior:

<u>switch example</u>	<u>if-else equivalent</u>
<pre>switch (x) { case 1: cout << "x is 1"; break; (x == 2) case 2: cout << "x is 2"; break; default: cout << "value of x unknown"; }</pre>	<pre>if (x == 1) { cout << "x is 1"; } else if { cout << "x is 2"; } else { cout << "value of x unknown"; }</pre>

No Break Statement Using

To understand what happens when no **Break** statement using, we'll trace through this version:

```
switch (ch)
{
  case 'a':
    ++aCnt;    // oops: should have a break statement
  case 'e':
    ++eCnt;    // oops: should have a break statement
  case 'i':
    ++iCnt;    // oops: should have a break statement
  case 'o':
    ++oCnt;    // oops: should have a break statement
  case 'u':
    ++uCnt;    // oops: should have a break statement
}
```

Assuming that value of **ch** is 'i'. Execution begins following case 'i' thus incrementing **iCnt**. Execution does **not stop** there but **continues** across the case labels incrementing **oCnt** and **uCnt** as well. If **ch** had been 'e', then **eCnt**, **iCnt**, **oCnt**, and **uCnt** would all be incremented. So, **Forgetting to provide a break is a common source of bugs in switch statements.**



Although it is not strictly necessary to specify a **break** statement after the last label of a switch, the safest course is to provide a break after every label, even the last. If an additional case label is added later, then the break is already in place.



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



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



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



The expression evaluated by a switch can be arbitrarily complex. In particular, the expression can define and initialize a variable:

```
switch(char ch = getchar())
```

In this case, **ch** is initialized, and the value of **ch** is compared with each case label. The variable **ch** exists throughout the entire switch statement but not outside it.



Case labels must be **constant integral expressions**. For example, the following labels result in compile-time errors:

```
// illegal case label values  
case 3.14:      // non-integer  
case ival:     // non-constant
```

It is also an error for any two case labels to have **the same value**. For example:

```
case 4:  
    cout<<"four\n";  
    break;  
case 4:  
    cout<<"again four\n";  
    break;
```




The following program shows a switch statement in action in a program that determines your average based on your grade:

```
#include "iostream.h"
void main()
{
    char grade;
    cout << "Enter your grade: ";
    cin >> grade;
    switch (grade)
    {
        case 'A':
            cout << "Your average must be between 90 – 100\n";
            break;
        case 'B':
            cout << "Your average must be between 80 – 89\n";
            break;
        case 'C':
            cout << "Your average must be between 70 – 79\n";
            break;
        case 'D':
            cout << "Your average must be between 60 – 69\n";
            break;
        default:
            cout << "Your average must be below 60\n";
    }
}
```



Modify the grade program, when you enter 'a' or 'A', the output must be "Your average must be between 90 - 100", and 'b' or 'B' , the output must be "Your average must be between 80 - 89", and so on. (by using switch statement only).



Exercise



Is there any difference between the two fragments below, and if there is no difference, what is the output when `ch='u','o','i','O',and'U'`.

```
int vowelCnt = 0;
// ...
switch (ch)
{
    // any occurrence of a,e,i,o,u increments vowelCnt
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        ++vowelCnt;
        break;
}
```

```
int vowelCnt = 0;
// ...
switch (ch)
{
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
}
```

Exercise



What is the output of the following fragments:

```
switch (x)
{
    case 1:
        cout<<"sat\n";
        break;
    case 2:
        cout<<"sun\n";
        break;
    case 3:
        cout<<"mon\n";
}
```

When `x=1, 4, 2, and 3`.



```
switch (v)
{
  case 'a':
    cout<<"alpha"
  case 'b':
    cout<<"beta";
  case 'c':
    cout<<"gama";
    break;
  default:
    cout<<"not on list"
}
```

When $v='a', 'b',$ and $'c'$.



Iteration Structures (loops)

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

The while loop

Its format is:

```
while (expression)
    statement;
```

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



// custom countdown using while

```
#include "iostream.h"
void main ()
{
    int n;
    cout << "Enter the starting number > ";
    cin >> n;
    while (n>0)
    {
        cout << n << ", ";
        --n;
    }
    cout << "FIRE!\n";
}
```

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



The **while loop** is using when, **the number of times is un-fixed**.



If we want more than a single statement to be repeated, we can specify a **block** using braces { }.



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



It is not unusual for a while loop to have an **empty body** (i.e., a null statement). The following loop, for example, sets **n** to its greatest odd factor.

```
while (n % 2 == 0 && n /= 2);
```

Here the loop condition provides all the necessary computation, so there is no real need for a body. The loop condition not only tests that **n** is even, it also divides **n** by **two** and ensures that the loop will terminate should **n** be zero.



When a while loop has a block, **don't Put a Semicolon after the Relational Expression**. For example, in the program below, **it causes an infinite loop!**

```
#include "iostream.h"
void main()
{
    float x,y;
    char ch=' ';
    while (ch != 'n'); // oops, Infinite Loop!
    {
        cin>>x>>y;
        if(y==0)
            cout<<"divisor can't be zero\n";
        else
            y=x/y;
        cout<<"do another y/n";
        cin>>ch;
    }
}
```

C++ compiler doesn't tell you about the above mistake during compilation time. So, be careful.



Remember, Assignment Has Low Precedence!

Inside a condition is another **common place** where **assignment** is used as a part of a larger expression. Writing an assignment in a condition can shorten programs and clarify the programmer's intent. For example, the following loop uses a function named **getchar()**, which returns **char** values. We can test those values until we obtain some desired value say, 'A':

```

char ch= getchar();
while (ch != 'A')
{
    //do something ...
    ch= getchar();
}
    
```

→

```

char ch;
while ((ch = getchar()) != 'A')
{
    //do something ...
}
    
```

Without the parentheses, the operands to **!=** would be the value returned from calling **getchar** and 'A'. The true or false result of that test would be assigned to **ch** clearly not what we intended!. So, **The additional parentheses around the assignment are necessary because assignment has lower precedence than inequality.**



When you use the function **getchar()** to get a character, you must use the library **"stdio.h"**, as following:

```
#include "stdio.h"
```



The **assignment** in the while loop represents a very common usage. Because such code is widespread, it is important to study this expression until its meaning is immediately clear.



// Example about using assignment as apart of the relational expression

```
#include "iostream.h"
#include "stdio.h"
void main()
{
    char ch;
    cout<<"to quit, input 'q'\n";
    while ((ch = getchar()) != 'q')
        cout<<ch<<"\n";
}
```

Exercise



What will happen if there is a semicolon after the relational expression of the above example?

Exercise



What is the output of the program below?

```
#include "iostream.h"
void main()
{
    int num = 0;
    while (num++ < 10)
        cout << num << " ";
}
```



The do .. while Statement

The **do .. while** statement (also called do loop) is **similar** to the **while** statement, **except that its body is executed first and then the loop condition is examined**. Its format is:

```
do
    statement;
while (condition);    // wow!!!, there is a semicolon
```

The **statement** in a **do** is executed before condition is evaluated. So, **the do .. while statement is granting at least one execution of statement even if condition is never fulfilled**. For example, the following example program echoes any number you enter until you enter **0**.



// number echoer

```
#include "iostream.h"
void main ()
{
    unsigned long n;
    do
    {
        cout << "Enter number (0 to end): ";
        cin >> n;
        cout << "You entered: " << n << "\n";
    } while (n != 0);
}
```



The **do .. while** loop is using when, **the number of times is un-fixed**.



The condition cannot be empty. If condition evaluates as **false**, then the loop **terminates**; otherwise, the loop is **repeated**.



If we want more than a single statement to be repeated, we can specify a **block** using braces { }.



Unlike a while statement, a do-while statement always ends with a **semicolon**.



The **do .. while loop** is less frequently used than the **while loop**. It is useful for situations where we need the loop body to be executed at least once, regardless of the loop condition. For example, suppose we wish to repeatedly read a value and print its square, and stop when the value is zero. This can be expressed as the following loop:

```
do
{
    cin >> n;
    cout << n * n << "\n";
} while (n != 0);
```

Unlike the **while loop**, the **do loop** is never used in situations where it would have a null body. Although a do loop with a null body would be equivalent to a similar while loop, the latter is always preferred for its superior readability.



// Example about do .. while, repeatedly asks user for pair of numbers to sum

```
#include "iostream.h"
void main()
{
    char ch;
    int val1, val2;
    do
    {
        cout << "please enter two values: ";
        cin >> val1 >> val2;
        cout << "The sum of " << val1 << " and " << val2
            << " = " << val1 + val2 << "\n\n";
        << "More? [yes,y][no,n] ";
        cin >> ch;
    } while(ch!='n');
}
```



The for Statement

The **for statement** (also called **for loop**) is similar to the **while statement**, but has two additional components: an expression which is evaluated only once before everything else, and an expression which is evaluated once at the end of each iteration. Its format is:

```
for (expression1; expression2; expression3)
    statement;
```

First **expression1 (initializer)** is evaluated. Each time round the loop, **expression2 (condition)** is evaluated. If the outcome is nonzero then **statement** is executed and **expression3 (subscript increment or decrement)** is evaluated. Otherwise, the loop is terminated.



//Example, countdown using a for loop

```
#include "iostream.h"
void main ()
{
    int n;
    for (n=10; n>0; n--)
        cout << n << ", ";
    cout << "FIRE!\n";
}
```



The **for loop** is using when, **the number of times is fixed or un-fixed**.



If we want more than a single statement to be repeated, we can specify a **block** using braces { }.



The **for loop** repeats the statements as long as the condition is **true**.



The most common use of **for loops** is for situations where a variable is **incremented or decremented** with every iteration of the loop. The following for loop, for example, calculates the sum of all integers from **1** to **n**.

```
sum = 0;
for (i = 1; i <= n; ++i)
    sum += i;
```

This is preferred to the **while-loop** version we saw earlier. In this example, **i** is usually called the **loop variable**.



C++ allows the **first expression** in a for loop to be a **variable definition**. In the above loop, for example, **i** can be defined inside the loop itself:

```
for (int i = 1; i <= n; ++i)
    sum += i;
```



Any of the three expressions in a for loop may be **empty**. For example, removing the first and the third expression gives us something **identical** to a **while loop**:

```
for ( ; i != 0; )    // is equivalent to: while (i != 0)
    something;    // something;
```

Removing all the expressions gives us an infinite loop. This loop's condition is assumed to be always true:

```
for (;;)    // infinite loop
    something;
```

For loops with multiple loop variables are not unusual. In such cases, the **comma operator** is used to separate their expressions:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
    something;
```

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



Don't Put a Semicolon at the end of for statement. For example, in the fragment below, it separates the **cout statement!**, so it will execute one time (prints one star).

```
For (int i=0; i<=5; i++);
    cout<<"*";
```

C++ compiler doesn't tell you about the above mistake during compilation time. So, be careful.



You would not intend an infinite loop in your code, but mistakes do happen. If it happens to you, don't panic. You can use the **CTRL-BREAK** keyboard combination to end the program. Knowing you have encountered an infinite loop, you then can correct the code error that caused it.



// the following program calculates the factorial of a number inputted by the user. A factorial is the product of all the positive integers from 1 to that number. For example, the factorial of 3 is $3 * 2 * 1$, which is 6, while the factorial of 5 is $5 * 4 * 3 * 2 * 1$, which is 120.

```
#include "iostream.h"
void main()
{
    int num, counter, total = 1;
    cout << "Enter a number: ";
    cin >> num;
    cout << "The factorial of " << num << " is ";
    for (int counter = 1; counter <= num; counter++)
        total *= counter;
    cout << total;
}
```



Write program to read group of numbers ended with 999, using for loop, the program should count a number of positive and negative number, a number of numbers that larger than 100, and a number of odd and even numbers.

**Exercise**

Explain each of the following loops. Correct any problems you detect.

- (a)

```
for (int a = 0, ix = 0; ix != 100 && a != 100; +ix, ++a)
{
    // do something
}
```
- (b)

```
for ( ; ; )
{
    //do something
}
```
- (c)

```
char ch;
for( ; ch=getchar() != '.' ; )
    putchar(ch);
```
- (d)

```
for (int ix = 0; ix != sz; ++ix)
{
    // do something
}
if (ix != sz)
    // ...
```
- (e)

```
int ix;
for (ix != sz; ++ix)
{
    //do something
}
```
- (f)

```
for (int ix = 0; ix != sz; ++ix, ++ sz)
{
    //do something
}
```



Nesting Loops

You can nest one for loop inside another, or one while loop inside another. You also can nest a while loop inside of a for loop, or a for loop inside of a while loop or do loop, and so on.

With respect to **nesting for loop**, the following program prints five rows of ten X characters:



```
//program prints five rows of ten X characters
```

```
#include "iostream.h"
void main()
{
    for (int x = 1; x <= 5; x++)
    {
        for (int y = 1; y <= 10; y++)
            cout << "X";
        cout << '\n';
    }
}
```

The for loop for (int x = 1; x <= 5; x++) is the outer for loop. The for loop for (int y = 1; y <= 10; y++) is the inner for loop.

With nested for loops, for each iteration of the outer for loop, the inner for loop goes through all its iterations. By analogy, in a clock, minutes are the outer loop, seconds the inner loop. In an hour, there are 60 iterations of minutes, but for each iteration of a minute, there are 60 iterations of seconds.



Nested for loops can be used to read and print rows and columns for tables (two-dimensional arrays)

Another example is a program prompts the user for the total number of salespersons as well as the number of sales per salespersons, and has the user input each sale of each salesperson, and then afterward displays the average sale for each salesperson. The number of iterations of the outer for loop will be the number of salespersons. The number of iterations of the inner for loop will be the number of sales per salesperson.



//program finds the average sale for each salesperson

```
#include "iostream.h"
void main()
{
    int persons, int numSales;
    cout << "Enter number of salespersons: ";
    cin >> persons;
    cout << "Enter number of sales per salesperson: ";
    cin >> numSales;
    for (int x = 1; x <= persons; x++)
    {
        int sale, total = 0;
        float average;
        for (int y = 1; y <= numSales; y++)
        {
            cout << "Enter sale " << y << " for salesperson "
                << x << ": ";
            cin >> sale;
            total += sale;
        }
        average = (float) total / numSales;
        cout << "Average sales for salesperson #" << x
            << " is " << average << "\n";
    }
}
```

With respect to **nesting while loops**, The following is a modification of that program that prints 5 rows of 10 X characters but using nested while loops.



//program prints five rows of ten X characters

```
#include "iostream.h"
void main()
{
    int x = 0;
    while (x++ < 5)
    {
        int y = 0;
        while (y++ < 5)
            cout << "X";
        cout << "\n";
    }
}
```



Since each loop has a predictable number of iterations, using nested for loops is somewhat simpler than using nested while loops. However, both work.

**Exercise**

Compare between for and while Loops.

Exercise

Compare between the do .. while and while Loops.



Write a program to print four strings, the end of each string is '*'.



Write a program which count the No. of digits, coma, newline in four lines, the end of each line is ';'.



Write a program to read chars, then change each char in small letter to capital letter and vice versa. The program must be finish when the char is '*'.

- For small letter to capital letter
ch=ch-'a'+'A'
- For capital letter to small letter
Ch=ch-'A'+'a';



Write a program which inputs an octal number and outputs its decimal equivalent. The following example illustrates the expected behavior of the program:

Input an octal number: 214
Octal(214) = Decimal(532)



Write a program which produces a simple multiplication table of the following format for integers in the range 1 to 9:

1 x 1 = 1
1 x 2 = 2
...
9 x 9 = 81



Jump statements:

The break statement

Using **break** we can leave a loop (while, do, or for) even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its natural end:



// break loop example

```
#include "iostream.h"
void main ()
{
    int n;
    for (n=10; n>0; n--)
    {
        cout << n << ", ";
        if (n==3)
        {
            cout << "countdown aborted!";
            break;
        }
    }
}
```

In addition to the loop structures **break** statement may appear inside **switch statement**. It causes a jump out of that structure, and hence terminates it.



A **break** statement only applies **to the loop or switch immediately enclosing it**. It is an **error** to use the break statement outside a loop or a switch. The **break** statement, is used, commonly within an **if** structure.



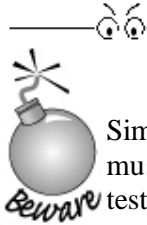
In **for loop**, if the **condition** is omitted, as the following:

```
for (int i = 0; ; ++i)
```

or if the **three parts** are omitted, as the following:

```
for ( ; ; )
```

It is essential that the body of the loop contain a **break** statement. Otherwise the loop will execute until it exhausts the system resources (**infinite loop**).



Similarly, if the **expression3 (subscript inc. or dec.)** is omitted, then the loop must exit through a **break** the loop body must arrange to change the value tested in the condition:

```
for (int i = 0; i != 10; )
{
    // body must change i or the loop won't terminate
}
```

If the body doesn't change the value of **i**, then **i** remains **0** and the test will always succeed (**infinite loop**).



In **nesting loops**, if you place a **break statement** in the inner loop, it will affect only that inner loop, and have no effect on the outer loop.

```

Outer loop
{
    //do something
    Inner loop
    {
        //do something
        if(condition)
        break;
    }
}

```

Stops the inner loop and the outer loop goes on

```

Outer loop
{
    //do something
    if(condition)
    break;
    Inner loop
    {
        //do something
    }
}
:
:
:

```

Stops the outer loop and so the inner loop stops too, and the rest of the program will execute

Exercise



Compare between the two programs and find the output of both.

```
#include "iostream.h"
void main()
{
    for (int x = 1; x <= 5; x++)
    {
        for (int y = 1; y <= 10; y++)
        {
            if (y%2==0)
                break;
            cout << "X";
        }
        cout << "\n";
    }
}
```

```
#include "iostream.h"
void main()
{
    for (int x = 1; x <= 5; x++)
    {
        if (x%2==0)
            break;
        for (int y = 1; y <= 10; y++)
            cout << "X";
        cout << "\n";
    }
}
```



Exercise



Is there any problem in the following fragment? If there is a problem what is the solution?

```
char ch;
while ((ch=getchar()) != '.')
{
    switch(ch)
    {
        case '+':
            float sum=0.0;
            for (int i=0; ; i++)
            {
                sum +=i
                if (sum == 100)
                    break;
                // ...
            }

        case '-':
            // ...

    } // end switch

} // end while
```



// program, Guess a number

```
#include "iostream.h"
void main()
{
    int num, counter, secret = 3;
    cout << "Guess a number between 1 and 10\n";
    cout << "You have 3 tries\n";
    for (int counter = 1; counter <= 3; counter++)
    {
        cout << "Enter the number now: ";
        cin >> num;
        if (num == secret)
        {
            cout << "You guessed the secret number!";
            break;
        }
    }
    cout << "Program over";
}
```



The continue statement

The **continue statement** causes the program to **skip or ignore the rest of the loop in the current iteration** as if the end of the statement block had been reached (**terminates the current iteration**), causing it to **jump to the start of the following iteration**. For example, we are going to skip the number 5 in our countdown:



// continue loop example

```
#include "iostream.h"
void main ()
{
    for (int n=10; n>0; n--)
    {
        if (n==5)
            continue;
        cout << n << ", ";
    }
    cout << "FIRE!\n";
}
```



continue statement applies to the loop immediately enclosing it. It is an **error** to use the continue statement outside a loop. The **continue** statement like the **break** statement, is used, commonly within an **if** structure. For example:



//Program which prints any character you entered, and ignores the dollar sign \$

```
#include "iostream.h"
#include "stdio.h"
void main ()
{
    char ch;
    for (int i=1 ; (ch=getchar()) != '\n'; i++)
    {
        if (ch=='$')
            continue;
        putchar();
        cout<< i <<"\n";
    }
}
```



In **while** and **do** loops, the next iteration commences from the **loop condition**.
 In a **for** loop, the next iteration commences from the **loop's third expression**.
 For example:

```
do ←
{
  cin>>x>>y;
  if (y==0)
  {
    cout<<"divisor can't be zero\n";
    continue;
  }
  y=x/y;
  cout<<"do another y/n";
  cin>>ch;
} while (ch != 'n');
```

Increment i
and check
condition
Ignored

```
while (ch != 'n') ←
{
  cin>>x>>y;
  if (y==0)
  {
    cout<<"divisor can't be zero\n";
    continue;
  }
  y=x/y;
  cout<<"do another y";
  cin>>ch;
}
```

Ignored

```
for (i=1; i<=5 ; i++) ←
{
  cin>>x;
  cout<<" i="<<i<<"\n";
  if (x==2)
  {
    continue;
  }
  x += 10;
}
```

Increment i
and check
condition
Ignored



In **nesting loops**, if you place a **continue** statement in the inner loop, it will affect only that inner loop, and have no effect on the outer loop. For example:

```
while (k <= more)
{
  for (i = 0; i < n; ++i)
  {
    cin >> num;
    if (num < 0)
      continue; // causes a jump to: ++i
    // process num here...
  }
  //etc...
}
```



Exercise



What is the output of the following fragment?

```
while ((ch=getchar())!='\n' )
{
    if (ch >= 'a' && ch <='z' || ch >= 'A' && ch <= 'Z')
        continue;
    for (int i=1 ; i<=10 ; i++)
    {
        if( i % 2 == 0)
            continue;
        cout<<ch<<" ";
    }
    cout<<"\n";
}
```

Exercise



Is the following program logical or not? If, it is logical, is it equivalent to the above fragment?

```
#include "iostream.h"
#include "stdio.h"
void main( )
{
    char ch;
    while (1)
    {
        ch=getchar( );
        if (ch=='\n')
            break;
        if (ch >= 'a' && ch <='z' || ch >= 'A' && ch <= 'Z')
            continue;
        for (int i=1 ; i<=10 ; i++)
        {
            if( i % 2 == 0)
                continue;
            cout<<ch<<" ";
        }
        cout<<"\n";
    }
}
```



Write program which read a sequence of 20 numbers and compute their sum, when $\text{sum} \geq 700$ the program must print the sum and stop even if not all the numbers are read, on the other hand if the number is zero, it must not be counted.



Write a program which counts the No. of chars, digits, and special chars in 20 lines.



Write a program which counts the No. of newline, ',', ';', '%' in four lines.



Write a program which read a sequence of integers and counts the No. of odd number, the program must be finished when the integer is zero.



Write a program which reads a sequence of 50 numbers and prints the prime number only.



Write a program which reads a sequence of integers and counts the No. of positive and negative integers, on the other hand, if the number is zero it must not be counted. The program must be finished when the number is 99.



Compound Data Types

Integer, char, and float are **single, simple** data type (consist of one cell only), but compound data type are **structured** (consist of a **number of cells** may be similar may be not) like **arrays, pointers, references, and the structure (record)**.

Arrays

An **array** consists of a set of **elements**, all of which are of the **same type** and are **arranged contiguously in memory**. In general, only the array itself has a symbolic name, not its elements. Each element is identified by an **index (or subscript)** which denotes the position of the element in the array. The number of elements in an array is called its **dimension**. The dimension of an array is **fixed** and predetermined; **it cannot be changed during program execution**.

Why Arrays?

Arrays are suitable for representing composite data which consist of **many similar, individual items**. Examples include: a list of names, a table of world cities and their current temperatures, or the monthly transactions for a bank account.



Because of an array consists of a **set of elements**, you have to use the **loops** for **accessing each one**. *Mostly*, **for loop** is used for this purpose, for accessing the elements of **one dimensional array**, you need to **one for loop**, for accessing the elements of **two dimensional array**, you need to **two for loops**.



Arrays keep data as long as the program in execution.



One Dimensional Array

One dimensional array has a **one dimension**. It also called **list** or **vector**.

One Dimensional Array Definition

It must be **defined** before it can be used. The syntax for defining an array is almost identical to the syntax for defining integers, characters, or other variables. For example, you would define an integer variable **x** as follows:


```
int x;
```


By contrast, you would define an **array x** of **three** elements:


```
int x[3];
```

This definition contains an **array of integers**. You instead could define an array of floats, characters in the following manner:

```
float GPA [5];  
char grades[7];
```

 The definition of both a single variable and an array of variables begins with the data type followed by a variable name and ending with a semicolon. The **only difference** between defining a variable that **holds a single value** and an array is that, when defining an array, the variable name is followed by a **number within square brackets**. **That number is the array's size declarator**.

 The purpose of the **size declarator** is to tell the computer how much memory to reserve. The size declarator, combined with the data type of the array, determines how much memory to reserve.

 The **size declarators** used in above examples was a **literal**. A literal is a value that is written exactly as it is meant to be interpreted. For example, the numbers 3, 5, 7 are literals. You may use a **constant** instead of a literal as a **size declarator (this way is more logical than the literal way) for example**:

```
#include "iostream.h"  
void main ()  
{  
    const int d = 3;  
    int x [d];  
    .  
    .  
}
```



The **size declarator can not be a variable**. The following program attempts, **unsuccessfully**, to use a variable **d** in declaring the size of an array:

```
#include "iostream.h"
void main ()
{
    int d;
    cout << "Enter the number of elements:";
    cin >> d;
    int x [d];
}
```

The result is a **compiler error**. The compiler will flag the declaration of the array (**int x[d]**) and complain that a constant expression was expected.



The **first index in an array is always 0**. There are no exceptions. **The last index in an array is always 1 less than the number of elements in the array**; again, with no exceptions. For example, If you were counting three numbers, starting at 1, the last element would be number 3. However, if you are starting at 0 instead of 1, then the last number would be 2, not 3.



By far, the most common causes of security problems are so-called "**buffer overflow**" bugs. These bugs occur when a **subscript is not checked and reference is made to an element outside the bounds of an array or other similar data structure**. For example:



```
//Example of Index out of range
#include "iostream.h"
void main ()
{
    int testi[3];
    int i;
    for (i = 0; i <100; i++)
        cin>> testi[i];
    for (i = 0; i <100; i++)
        cout<< testi[i] << "\n";
}
```

Nothing stops a programmer from stepping across an array boundary except **attention to detail and thorough testing of the code**. It is not inconceivable for a program to compile and execute and still be fatally wrong. In other words, **C++ compiler doesn't tell you about this error, so be careful**.



One Dimensional Array Initialization

There are two methods of initializing an array. The first is **explicit array sizing**, in which the square brackets contain a numerical constant that explicitly specifies the size of the array. The second is **implicit array sizing**, in which the square brackets are empty and the size of the array is indicated implicitly by the number of elements on the right side of the assignment operator.



The syntax of initialization, with both explicit and implicit array sizing, is that the array declaration, is followed by an assignment operator and then, enclosed in curly braces, the values to be assigned to each array element, in order, are separated by commas.

Explicit Array Sizing

The following are examples of explicit array sizing:

```
int testi[3] = { 74, 87, 91 };  
float testf[4] = { 44.4, 22.3, 11.6, 33.3};  
char grades[5] = {'A', 'B', 'C', 'D', 'F' };  
string days[7] = {"Sunday", "Monday", "Tuesday", "Wednesday",  
"Thursday", "Friday", "Saturday"};
```



You do not have to assign values to each element of the array; the number of elements on the right-hand side of the assignment operator may be less than the number within the square brackets:

```
float testf[4] = { 44.4, 22.3, 11.6};
```

If you do not initialize all of the elements of an array, the uninitialized elements have a default value that depends on the data type of the array. For example, the default value is **0** for an integer array, **0.0** for a float array, and the null character, **'\0'**, for a character array.



The number of elements on the right-hand side of the assignment operator cannot be greater than the number within the square brackets.

Thus, the following statement will not compile, the error message being “too many initializers.”

```
float testf[4] = { 44.4, 22.3, 11.6, 33.3, 7.4}; // won't compile
```



If you leave an element uninitialized, all elements that follow it must be uninitialized. You can't, for example, alternate initializing and not initializing array elements. For example, the following statement won't compile:

```
float testf[4] = { 44.4, , 11.6, 33.3}; // won't compile
```

Implicit Array Sizing

The following are examples of implicit array sizing:

```
int testi[ ] = { 74, 87, 91 };
```

```
float testf[ ] = { 44.4, 22.3, 11.6, 33.3};
```

```
char grades[ ] = {'A', 'B', 'C', 'D', 'F'};
```

```
string days[7] = {"Sunday", "Monday", "Tuesday", "Wednesday",  
"Thursday", "Friday", "Saturday"};
```

The first array, **testi**, allocates memory for **three integers**. Since the square brackets are blank, the compiler allocates memory based on the number of elements to the right side of the assignment statement, and so on for the rest.



You cannot have both empty square brackets and no initialization, as in the following example:

```
int testi[ ];
```

The compiler error message will be that the array is of unknown size. This of course is a problem since the computer has no way of knowing how much memory to allocate for the array.



You can initialize a character array using the same syntax as you would to initialize an array of another data type such as an integer or a float. However, there are some important differences between character arrays and arrays of numeric data types. For example:

The following two initializations of a character array to **name** are different in syntax but identical in effect:

```
char name[ ] = {'J', 'e', 'f', 'f', '\0' };
char name [ ]= "Jeff";
```

The character ‘\0’ is the escape sequence for a null character. The **0** in ‘\0’ is a zero, not a big letter **o**. The zero corresponds to the ASCII value of the null character. **The null character signals cout when to end the output of a character array.** For example, the following program outputs, as expected, “Jeff”:

```
#include "iostream.h"
void main ()
{
    char name[ ] = {'J', 'e', 'f', 'f', '\0' };
    cout << name;
}
```

The result would be the same if the alternate syntax of `char name = “Jeff”` was used to initialize the character array.

By contrast, the following program outputs “Jeff!!!!+ ?.”

```
#include "iostream.h"
void main ()
{
    char name[ ] = {'J', 'e', 'f', 'f'};
    cout << name;
}
```

The strange characters after “Jeff” (which may differ when you run the program) sometimes are referred to as “garbage characters”;

Finally, The latter syntax (`char name = "Jeff";`) usually is preferred by programmers simply because it is easier to type and it is automatically inserting a null character ‘\0’ as the end of the array.



It is important to remember the null-character when initializing an array of characters to a literal. For example, the following is a compile-time error:

```
char ch3[6] = "Daniel"; // error: Daniel is 7 elements
```

While the literal contains only six explicit characters, the required array size is seven, six to hold the literal and one for the null.



No Array Copy or Assignment. It is not possible to initialize an array as a copy of another array. Nor is it legal to assign one array to another:

```
void main()
{
    int ia[] = {0, 1, 2};
    const int size = 3;
    int ia3[size];    // ok: but elements are uninitialized!
    ia3 = ia;        // error: cannot assign one array to another
}
```



You can create arrays that are constants. For example, the following array contains the number of days in each month (for February, we assume a non-leap year).

```
const int daysInMonth [ ] = { 31, 28, 31, 30, 31, 30,
                              31, 31, 30, 31, 30, 31 };
```

You must use initialization when creating a constant array, just as you must use initialization when creating a constant variable.



//Count capitals characters and change them to small.

```
#include "iostream.h"
void main()
{
    int i;
    int count=0;
    const int size=10;
    char letters[size];
    for ( i=0 ; i < size ; i++)
    {
        cin>>letters[i];
        if ( letters[i] >= 'A' && letters[i] <= 'Z' )
        {
            count++;
            letters[i]= letters[i] - 'A' + 'a' ;
        }
    }
    cout<<"number of capitals = "<<count<<"\n";
    cout<<" array elements after changing\n";
    for ( i=0 ; i<size ; i++)
        cout<< letters[i] << "\n";
}
```



// Compare between two arrays of integers and find whether they are equivalent or not.

```
#include "iostream.h"
void main()
{
    int i;
    int flag=1;
    int a[10],b[10];
    for (i=0; i<10 ; i++)
        cin>> a[i];
    for (i=0; i<10 ; i++)
        cin>> b[i];
    for (i=0; i<10 ; i++)
        if(a[i]!=b[i])
        {
            flag=0;
            break;
        }
    if (flag)
        cout<<" a and b are equivalent\n";
    else
        cout<<" a and b are not equivalent\n";
}
```



//Convert a char string of digits into its numeric value.

```
#include "iostream.h"
void main()
{
    int i, n;
    char s[5];
    for (i=0; (s[i]=getchar())!='\n'; i++);
    s[i]='\0';
    i=n=0;
    while (s[i]!='\0')
        n=n*10+s[i++]-'0';
    cout<<"the numeric value = "<<n<<"\n";
}
```

Exercise



Which, if any, of the following definitions are in error?

- (a) `int ia[7] = { 0, 1, 1, 2, 3, 5, 8 };`
- (b) `int ivec = { 0, 1, 1, 2, 3, 5, 8 };`
- (c) `int ia2[] = ia1;`
- (d) `int ia3[] = ivec;`
- (e) `char ch3[6] = "Daniel";`
- (f) `char mystext = "Hello";`

Exercise



How can you initialize some or all the elements of an array?

Exercise



What is the fatal wrong that may be happen when using arrays?



Write program to read 15 integer number then checkout whether the No. 70 is found or not.



Write program to read 20 chars and count the number of small vowel letters, then change them to capital.



Strings

A C++ **string** is simply **an array of characters**. For example,

```
char str[ ] = "HELLO"; // string
```

defines **str** to be an array of six characters: five letters and a null character. The terminating null character `'\0'` is inserted by the compiler. By contrast,

```
char str[ ] = {'H', 'E', 'L', 'L', 'O'}; // array of chars
```

Why a null character ('\0')?

Strings are in fact sequences of characters, For example, the following array:

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

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

jenny

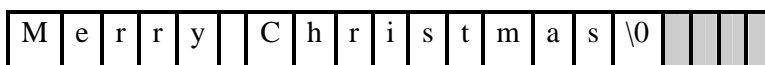


Therefore, in this array, in theory, we can store sequences of characters up to 20 characters long. But we can also store shorter sequences. For example, jenny could store at some point in a program either the sequence "Hello" or the sequence "Merry Christmas", since both are shorter than 20 characters.

Therefore, since the array of characters can store shorter sequences than its total length, a special character is used to signal the end of the valid sequence: the null character, whose literal constant can be written as `'\0'` (backslash, zero).

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

jenny



Notice how after the valid content a null character (`'\0'`) has been included in order to indicate the end of the sequence.



C Library String Functions

The Standard C library provides a set of functions that **operate on C-style strings**. To use these functions, we must include the associated C header file (**string.h**).

Name	Function
gets(s)	Read a string s from the keyboard.
puts(s)	Prints the string s on the screen.
strlen(s)	Returns the length of s, not counting the null.
strcmp(s1, s2)	Compares s1 and s2 for equality. Returns 0 if s1 == s2, positive value if s1 > s2, negative value if s1 < s2.
strcat(s1, s2)	Appends s2 (source) to s1 (target). Returns s1.
strcpy(s1, s2)	Copies s2 (source) into s1 (target). Returns s1.
strncat(s1, s2, n)	Appends n characters from s2 (source) onto s1 (target). Returns s1.
strncpy(s1, s2, n)	Copies n characters from s2 (source) into s1 (target). Returns s1.



gets(s) function is automatically inserting a null character '\0' as the end of the string.



We can use **cin** to get strings with the extraction operator (>>) as we do with fundamental data type variables:

```
cin >> mystring;
```

However, as it has been said, **cin** extraction **stops reading as soon as it finds any blank space character**, so in this case we will be able to get just one word for each extraction. This behavior may or may not be what we want; for example if we want to get a sentence from the user likes:

```
Blank space  _____↑
                Law is a bottomless pit
```

Output: Law

This extraction operation would not be useful. So, it is better to use the function **gets()**.



It is better to call **strlen** function by using an **integer variable** as the following:

```
len=strlen(str);
```

Length of **str** is saved in the variable **len**.



You cannot use **relational operators** to compare the value of one string to another, such as in the following code fragment:

```
char str1[80] = "Devvie Kent";
char str2[80] = "Devvie Kent";
if (str1 == str2)
    cout << "The two strings are equal";
else
    cout << "The two strings are not equal";
```

The output will always be: “**The two strings are not equal.**” The reason is that the value of each array name is the base address of that array. Thus, the comparison is not of values, but of addresses. Two variables cannot have the same address, so the result of the comparison for equality will be false. SO, you must use the **strcmp** function to compare the values of two strings.



Mostly, **strcmp** function is used with **if statement**. For example, we want to check whether **s1** is equal to “**quit**” or not:

```
if( !(strcmp(s1,"quit")))
```

Or

```
if( strcmp(s1,"quit") == 0)
```

Both, are right. So when you use **strcmp** function without decision structure, it will become **meaningless**.



The **strcmp** function takes two arguments. Both can be **string literals** or one of them or neither one.



The **strcat** function takes two arguments. The first argument cannot be a **string literal** since a value is being assigned to it. The second argument, the source string, may be a **character array**, or a **string literal**. Also, you need to make sure that the size of **s1 (target)** must **larger** than the **s2 (source)**, else you will make a fatal wrong, for example:

```
char target[80] = "Jeff";
char source[40]= " Kent";
strcat(target, source);
cout << target; // outputs "Jeff Kent"
```



You cannot assign the value of one string to another with an **assignment operator (=)**, such as in the following code fragment:

```
char target[80] = "Jeff Kent";
char source[80] = "Micaela";
target = source;
```

The value of `source` is the base address of its array. Thus, the assignment operator assigns the address of `source` to `target`, not the value of `source` to `target`.

So, you must use the **strcpy** function to assign the value of one string (the source string) to another string (the target string). The **strcpy** function takes two arguments. The first argument cannot be a **string literal** since a value is being assigned to it. The second argument, the source string, may be a **character array**, or a **string literal**. The following code fragment illustrates the use of the **strcpy** function:

```
char target[80] = "Jeff Kent";
char source[80] = "Micaela";
strcpy(target, source);
```

You need to be careful when using the **strcpy** function that the **source string is not larger than the target string**.



Although C++ supports C-style strings, they should not be used by C++ programs. C-style strings are a surprisingly rich source of bugs and are the root cause of many, many security problems.



// Program which read two strings, and stop when the first string is "quit", when each two strings are read, the program must print the smallest one and its length.

```
#include "iostream.h"
#include "string.h"
#include "stdio.h"
void main()
{
    char s1[10], s2[10];
    int n;
    for(;;)
    {
        gets(s1);
        if( !(strcmp(s1,"quit")))
            break;
        gets(s2);

        if( strcmp(s1,"quit") < 0)
        {
            puts(s1);
            n = strlen(s1);
            cout<<"Length = "<<n<<"\n";
        }
        else
        {
            puts(s2);
            n = strlen(s2);
            cout<<"Length = "<<n<<"\n";
        }
    }
    cout<<"Finish\n";
}
```

Exercise



Explain the differences between **strcpy** and **strncpy**. What are the advantages of each? The disadvantages?



Write program which read two strings s1 and s2 and concatenating them in s2.



Write program which read a string and print it in reverse order.



Write program to convert an integer number (numeric value) to string of digits.



Write program which read a string contains a number followed by an operation then another number (e.g 25*3) the program must perform the operation between the two numbers then print the result.



Multidimensional Arrays

An array may have more than one dimension (i.e., two, three, or higher). As before, elements are accessed by indexing the array. A **separate index** is needed for each dimension.

Two Dimensional Array Definition

It must be **defined** before it can be used. The syntax for defining two dimensional array is as the following:

```
int a[3][4]; // array of integers consists of three rows and four columns
float ff[6][6]; // array of reals consists of six rows and six columns
char name[10][10]; //Equivalent to array of 10 strings
```



The **size declarator** cases with two dimensional array are as in one dimensional array, but the difference, **two size declarators are needed**, one for rows and the second for columns, for example:

```
const int rows=10;
const int columns=5;
int b [rows] [columns];
```



Remember that array **indices** always begin by **zero**.



Processing a two dimensional array is similar to a one-dimensional array, **but uses nested loops instead of a single loop. Mostly, two indices are needed, one for rows and the second for columns.**



As in one dimensional array, you need to make sure that **indices or subscripts are not outside the bounds of an array dimensions**. Else you will make a fatal wrong.



Two Dimensional array Initializing

The array may be initialized using a **nested initializer** (specifying bracketed values for each row):

```
int seasonTemp[3][4] = {
    {26, 34, 22, 17},
    {24, 32, 19, 13},
    {28, 38, 25, 20}
};
```

Because this is mapped to a one-dimensional array of **12** elements in memory, it is equivalent to:

```
int seasonTemp[3][4] = { 26, 34, 22, 17, 24, 32, 19, 13, 28, 38, 25, 20};
```



The **nested initializer** is preferred because as well as being more informative, it is more versatile. For example, it makes it possible to initialize only the first element of each row and have the rest default to zero:

```
int seasonTemp[3][4] = {{26}, {24}, {28}};
```

With respect to **character array**, the following two initializations of a **name** are different in syntax but identical in effect:

```
char name[3][20] = {
    {'A', 'l', 'i', '\0'},
    {'A', 'h', 'm', 'e', 'd', '\0'},
    {'M', 'a', 'h', 'm', 'm', 'o', 'd', '\0'}
};
```

```
char name[3][20] = {"Ali", "Ahmed", "Mahmmod"};
```



The latter syntax (`char name[3][20] = {"Ali", "Ahmed", "Mahmmod"};`) usually is preferred by programmers simply because it is easier to type and it is automatically inserting a null character `'\0'` as the end of the string.



We can also **omit the first dimension (but not subsequent dimensions)** and let it be derived from the initializer:

```
int seasonTemp[ ][4] = {
    {26, 34, 22, 17},
    {24, 32, 19, 13},
    {28, 38, 25, 20}
};
```

Columns dimension can't be empty

Exercise



Why the following initialization is invalid? What do you think?

```
int seasonTemp[3][ ] = {
    {26, 34, 22, 17},
    {24, 32, 19, 13},
    {28, 38, 25, 20}
};
```



// Program which reads a two dim. array and prints the squares of its elements.

```
#include "iostream.h"
void main()
{
    int b[5][4];
    int i, j;
    for( i=0; i<5 ; i++ )
    for( j=0; j<4 ; j++ )
        cin>> b[i][j];
    for( i=0; i<5 ; i++ )
    {
        for( j=0; j<4 ; j++ )
            cout<< b[i][j] * b[i][j];
        cout<< "\n";
    }
}
```



// Program which reads a **square array c(4,4)** and finds the sum of its main and secondary diagonal.

```
#include "iostream.h"
void main()
{
    int c[4][4];
    int i, j, s, s1;
    for( i=0; i<4 ; i++ )
    for( j=0; j<4 ; j++ )
        cin>> b[i][j];
    s=s1=0;
    for( i=0; i<4 ; i++ )
    for( j=0; j<4 ; j++ )
        if(i==j)
            s += c[i][j];
        else if (i+j ==3)
            s1 += c[i][j];
    cout<<" the sum of main diagonal = "<<s<<"\n";
    cout<<" the sum of secondary diagonal = "<<s1<<"\n";
}
```



// Program which reads the names of 10 students and sorts them in ascending order.

```
#include "iostream.h"
#include "string.h"
#include "stdio.h"
void main()
{
    char name[10][15], temp[15];
    int i, j;
    for( i=0; i<10 ; i++ )
        gets(name[i]);
    for( i=0; i<9 ; i++ )
    for( j=i+1; j<10 ; j++ )
        if ( strcmp(name[i], name[j]) > 0 )
        {
            strcpy(temp, name[i]);
            strcpy(name[i], name[j]);
            strcpy(name[j], temp);
        }
    cout<<"\n Names after sorting \n";
    for( i=0; i<10 ; i++ )
        puts(name[i]);
}
```



Write a program to multiply two arrays: a (n,n) and b (n,n).



Write a program to read the names of 20 students and sorts them in descending order.



Write a program to read A (3,4) and print it column by column.



Write a program to read a square array c (4,4), then change the values of the triangles placed up and under the main diagonal to **(-1)** and **(0)** respectively.



References

1. **Sharam Hekmat, "C++ Programming", 1998.**
2. **Jeff Kent, "C++ Demystified: A Self-Teaching Guide", 2004.**
3. **Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo, " C++ Primer", fourth edition, 2005.**
4. **Juan Soulié, "C++ Language Tutorial", 2008.**