University of Salahaddin

College of Education

Department of Mathematic

Fourth Year

programming Language C++

Theory & Practical

Academic Year 2017-2018

Dr. Salar Mustafa Majeed

Introduction to programming languages

What is a programming language?

A computer program is a series of simple instructions that tell your computer to do some particular task. A programming language makes it easier and efficient for people to communicate with computers. Computers don't speak English. They are so simple that the only things they understand are ones (1s) and zeros (0s) like 0100 0100. Does this mean anything to you? All programs look like the example above, and collections of numbers, known as binary numbers because only digits involved are 0 and 1.

There are many different types for computer languages, each developed for a specific kind of use, and each language encourage a different approach.

A Brief History of Programming Language

In the early days, computers were programmed using instructions that the hardware understood directly. From machine language, to human-readable assembly language, to high lever language which abstract away from the details of the hard ware, hence become portable to different type of hardware, the programming language at its third generation, start boom rapidly. The most famous early imperative high-level languages are FORTRAN, COBOL AND ALGOL 60. The Main focus is machine state or the set of values stored in memory location, and command-driven.

FORTRAN (FORmula TRANslator)

FORTRAN was the first effectively implemented high-level language, it pioneered many techniques of scanning, parsing, register allocation, code generation, and optimization, it introduced variables, loops, procedures, statement labels and much more.

- Developed by John Backus;
- Fortran I (1957), designed for IBM 704 computer, all control structures corresponded to 704 machine instructions;
- First manual 1956, first successful compiler 1958;
- Fortran II (1958), independent compilation and fix the bugs;
- Fortran IV (1960-1962), explicit type declarations, logical selection statement, and subprogram names could be parameter;
- Fortran 77 (1977), able to hand character string, logical loop control statement and IF-THEN-ELSE statement;
- Fortran 90 (1990), uses modules, dynamic arrays, pointer, recursion CASE statement and parameter type checking.

COBOL – 1960

It is the first business-oriented computation language lead by Department of Defense personal, Grace Hopper, designed for business application, features for structuring data and managing files. *COBOL is still the most widely used business application language, used to be very popular in business and government.*

- First macro facility in a high-level language, very strict program organization;
- Hierarchical data structures (records), data structures, record type introduced for the first time;
- Nested selection statements, though poor control structures;
- Long names (up to 30 characters);
- English name for arithmetic operators;
- Data and code were in completely separated sections;
- Verbs were first word in every statement.

ALGOL (Algorithmic Language)

ALGOL was an international effort to design a universal language and never popular, but virtually all languages after 1958 used ideas pioneered by the Algol designs for it introduced and formalized many common language features of today. Realized the importance of standardization, and FORTRAN was one company's product (IBM), ALGOL was developed by a joint European-American committee. Many concepts were introduced: Language definition; Structured; Arrays can have variable bounds at compile time; Contained several structured control statements.

ALGOL Goals

- Should be as close as possible to standard math notation;
- Readable without too much additional explanation or even comments;
- Can be used for describing computing processes (algorithms) in publications;
- Mechanically translatable into machine code.

Advances in high-level languages

The 1960 - 1970s brought many new languages, building on experiences with the early ones. **Basic** is the first in history language of personal computing. **Pascal** is an ALGOL-like imperative language that gained a vast following primarily due to its utility for teaching programming. Another notable language of this time is **Simula** which is the first mainstream language to have object-oriented features.

BASIC (1964)

- Designed by Kemeny & Kurtz at Dartmouth;
- designed for beginners, unstructured but popular on microcomputers in 70's ;
- The first programming language for many programmers: designed to be easy to learn;
- Very simple, limited, though still general-purpose;
- Current popular dialects: QuickBasic and Visual BASIC;
- Present-day versions of Basic are full-fledged languages—not "basic", and not easy to learn any more.

Pascal (1971)

- Developed by Niklaus Wirth in the late 60's and early 70's; built on his earlier work on Algol W;
- Small, simple, nothing really new. A conceptually simplified and cleaned-up successor of Algol 60;
- Inclusion of features that encourage well-written and well-structured program;
- A great language for teaching structured programming. Became the primary teaching language in the world by mid-80's;
- Data types are a prominent feature of Pascal; integer, real, Boolean, char;
- Structured user-defined types include, arrays, records, sets and file;

Simula (Simulation Languages)

- Developed in the 60's at the Norwegian Computing Centre in Norway by Nygaard and Dahl;
- Designed primarily for simulating discrete systems, including concurrent processes;
- Simula '67 was the best known version;
- Based on Algol '60 with one very important addition, the CLASS concept;
- Primary Contribution -- Introduced the central concepts of object orientation: classes and encapsulation.
- Predecessor of Smalltalk and C++.

Smalltalk (1972-1980)

- Developed at Xerox PARC, initially by Alan Kay, later by Adele Goldberg;
- Inspired by Simula, Sketchpad, Logo, cellular biology;
- First full implementation of an object-oriented language;
- It is the purest object-oriented language ever designed (till now), cleaner than Java, much cleaner than C^{++} ;
- Comes complete with a graphical interface and an integrated programming environment;
- In skilled hands, a powerful tool.

C (1972)

C is arguably the most widely used imperative language today. Various languages have branched away from C by adding object-oriented features. The most obvious are C++ and Java.

- Designed for systems programming at Bell Labs by Dennis Ritchie;
- Evolved primarily from BCPL, B, and also ALGOL 68;
- A great tool for systems programming and a software development language on personal computers;
- Powerful set of operators, but poor type checking;
- Initially spread through UNIX;

- Once fashionable, still in use, but usually superseded by C++;
- Dangerous if not used properly: not recommended to novice programmers;
- Relatively low-level.

C++ (1980)

C ++ is an object-oriented extension of the imperative language C. This is a hybrid design, with object orientation added to a completely different base language.

- Developed at Bell Labs by Bjarne Stroustrup;
- Evolved from C and SIMULA 67;
- Facilities for object-oriented programming, mostly from SIMULA 67, were added to C;
- Also has exception handling (from Ada) ;
- A large and complex language, Complicated syntax, difficult semantics, in part because it supports both procedural and OO programming;
- Very fashionable, very much in demand , rapidly grew in popularity, along with OOP;

Java (1995)

Java is a neat, cleaned up, sized-down reworking of C++ with full object orientation features.

- Developed by James Gosling from Sun Microsystems;
- 1991: Oak: a language for ubiquitous computers in networked consumer technology;
- Based on C++, but significantly smaller and simpler ; supports only OOP;
- Has references, but not pointers;
- 1995: renamed Java, retargeted for the Web
    - Incorporated into web browsers
    - Platform-independent active content for web pages
- Java also has:
    - Garbage collection (ex Lisp)
    - Concurrency (ex Mesa)
    - Packages (ex Modula)
- Designed for network & distributed programming, multi-tier applications (databases)

C# (2000)

C# is a simple, modern, general-purpose, object-oriented programming language developed by Microsoft within its .NET initiative led by Anders Hejlsberg and his team, wanted to build a new programming language that would help to write class libraries in a .NET framework. It made its appearance in 2000.

C# constructs closely follow traditional high-level languages C and C++ and being an object-oriented programming language, it has strong resemblance with Java, it has numerous strong programming features that make it endearing to multitude of programmers worldwide.

The following reasons make C# a widely used professional language:

- Modern, general-purpose programming language
- Object oriented.
- Component oriented.
- Easy to learn.
- Structured language.
- It produces efficient programs.
- It can be compiled on a variety of computer platforms.
- Part of .Net Framework.

Why so many language?

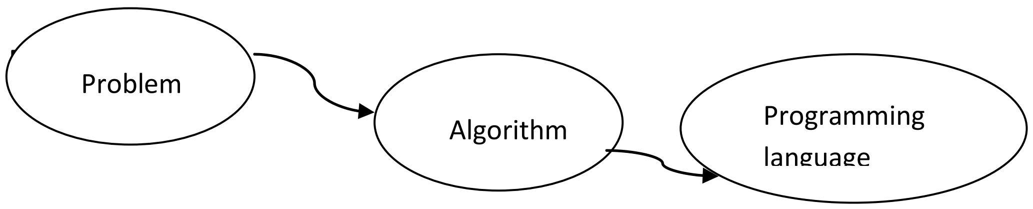
- Application domains;
- Levels of language;
- Different philosophies to solve problem;
- Language improvement;
- To make it easier to learn a new language.

What makes a good language?

- Clarity and simplicity;
- Ease of program verification.

Programming process

The programming process can be considered to be done in two steps. The first step is the step that invents the algorithm to solve the problem. The second step is step that decides the data structure on the basis of algorithm and writs a program using the programming language. However, in fact what kind of language is best to implement this algorithm is not solved.



There are several different types of statements, which can be categorized into several groups including: input, output, assignment condition and loop.

Input: A request to have the CPU accept information from the keyboard or other input device.

Output: A request to have the CPU display information on the screen or other output device.

Assignment: A request to have the CPU store or assign a value to a location in memory. The value may be the result of a calculation. The variable on the left side is where we wish the result of a calculation to be placed. The expression on the right represents the calculation. Expressions may contain variables and constants. The order of precedence for the operators of an expression is the same as you learned in algebra. If parentheses are not used to indicate an operator order, multiplication and division are done first, followed by addition and subtraction. If an expression has arithmetic operators at the same precedence, they are simply done from left to right.

Condition: A request to have the CPU choose between several different blocks of statements based upon a comparison or test of memory values.

Loop: A request to have the CPU repeats a block of statements multiple.

Algorithm

The algorithms are expressed by using natural language and numeral formulas. Writing algorithms in a specific programming language is the main portion of the study of programming.

An algorithm is the statement of the methodology used to solve a problem. A program is the implementation of that algorithm.

How it works Y=Y +1

This operation, increments the value of Y by 1. This means that 1 is added to the value of Y (Y=1), which up till now was equal to 2 is then stored is the variable Y.

Example

Find the highest number in the list of random numbers.

- The first number is the largest number in the list you've seen so far.
- Look at the next number and compare it with the largest number you've seen so far.
- If the next number is larger, then make that the new largest number you've seen so far.
- Repeat steps 2 and 3 until you have gone through the whole list.

Algorithm Largest Number

Input: A non-empty list of numbers L.

Output: The largest number in the list L.

Largest ← L_0

For each item in the list, do

"← " is a loose shorthand for changes to. For instance, "Largest ← item" means that the value of largest changes to the value of item.

" return" terminates the algorithm and outputs the value that follows

Flowchart

A flowchart is a diagrammatic representation that illustrates the sequence of operations to be performed to get the solution of a problem. One the flowchart is drawn, it become easy to write the program in any high level language.

Guidelines for drawing a flowchart

Flowcharts are usually drawn using some standard symbols, which are shown in Figure 1.

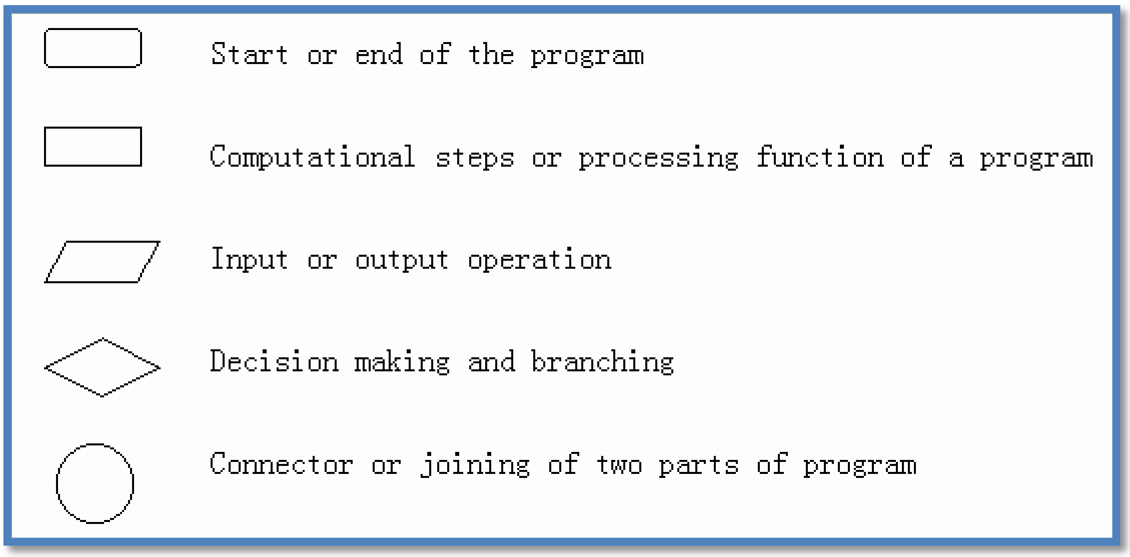
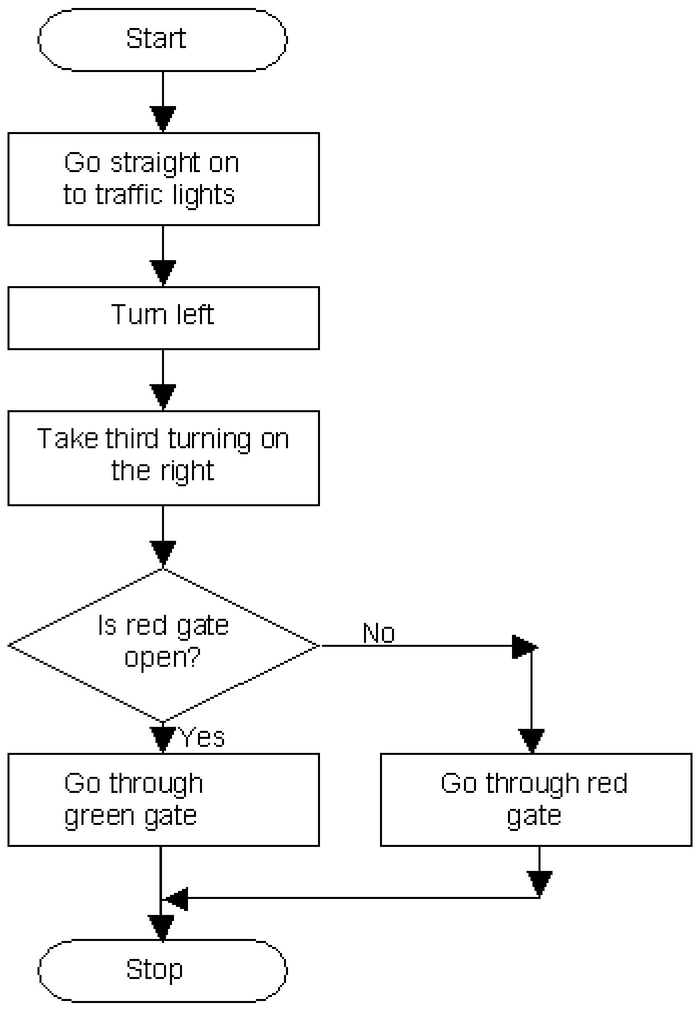

Start or end of the program
Computational steps or processing function of a program
Input or output operation
Decision making and branching
Connector or joining of two parts of program

Figure 1.1 Illustrate some standard symbols.


Start
Go straight on to traffic lights
Turn left
Take third turning on the right
Is red gate open?
No
Yes
Go through green gate
Go through red gate
Stop

Elements of C++

Variable declarations, arithmetic expressions, input, output and control statements are basic C++ constructions.

Comment Lines

Lines which contain comments, it doesn't a ctually affect the program's running at all. The compiler will ignore all characters from // to the end of the line. It's purpose is to explain in normal language what the programming is doing.

Header Files and # include Directives

Header files as the name suggest are the head of the program and are written outside the main body. These files contain commands to run codes like cin>>, cout<<, getch (); clrscr (). For example,

#include<**iostream.h**> : The main header file has the commands to run commands like clrscr(); and getch();

#include<**conio.h**> : This header file has codes to run cin>> & cout<<.

# include <**math.h**> : This header file contains information about mathematical function, such as sqrt(), sin and cos.

Note that clrscr(); means Clear the screen if this does not given then the output of previously

opened will also come along the output of your current program output.
getch(); is the command to give a pause to the console output screen , if not entered then the

output will be shown but if you will enter any value the screen will return to the coding window.

To us getch() you need to include the conio.h header file.

The main() function

```
main()
 {
   Statements;
  }
```

Every C++ program must define main(). Any text that is written between the { and }symbols forms the function body.

Values, types and constants

C++ classifies data values into types according to how they are stored in memory and what operations can be carried out on them. Types whose values represent numbers are called arithmetic types. The arithmetic types are divided into integer types, whose values represent whole numbers, and floating types, whose values can contain decimal points.

The term constant is used for constant data objects- data objects whose values cannot be changed. Such a constant is declared by prefixing the declaration with the keyword const. A constant is given a value when it is declared, and that value cannot be later changed by an assignment statement.

Declaration

The common term variable for data objects whose stored values can be changed during the execution of the program. Variables must be declared before they are used, by giving its type and name. For example,

int	integer
float	single precision floating point
char	character
string	is a sequence of characters
const	constant

Input and output

In C++, input is read from and output is written to streams. When iostream.h is include in a program, several standard streams are defined automatically. The stream cin is used for input, which is normally read from the user's keyboard. The stream cout is used for output, which is normally sent to the user's display. For example,

```
Cin>> x;        (input x)
Cout<< y;       (output y)
```

Note:

" \n " and endl : start printing on a newline.

" \t " : horizontal tab. Printer to jump to a predefind tab stop.

" \v " : vertical tab

Example

```
# include <iostream.h>

# include <conio.h>

int x,y ;


main()

{

x=7;

y=9;

cout<<"x="<<x<<"y="<<y<<endl;

cout<<"x="<<x<<endl<<"y="<<y<<endl;

cout<<x<<"\t"<<y;

getch();

}
```

display on the screen as the following

```
  x=4y=9

   x=4

  Y=9

  7              9
```

Arithmetic Operators

The C++ arithemetic operators are shown in Table 1.1.

Operator symbol	**Operation**
+	**Addition**
-	**Subtraction**
*****	**Multiplication**
/	**Division**
%	**The remainder after division**

The function pow(x,y) calculates the value of x raised to the yth power.

Example

```
# include <iostream.h>

# include <conio.h>

main()

{

cout <<5/3<< endl;

cout<<5./3<<endl;

cout<<4/2*2;

cout<<4/(2*2);

 getch();

}
```

display on the screen as the following

Relational Operators

Relational operators are used to compare tow values. The result of the comparison is either true(1) or false(0). Table 1.2 lists the various relational operators.

Table 1.2 Relational Operators

Operator	Meaning
>	greater than
<	less than
= =	equal to
!=	not equal to
>=	greater than or equal to
<=	less than or equal to
&&	logical **and** operator
||	logical **or** operator

The Compound assignment operators

If op= is a generalized denotation for the compound assignment operators, then the semantics of the expression:

variable op= expression

is specified by:

variable = variable op (expression)

For example sum/=2+4 is equivalent to: sum= sum/(2+4)

f=i*=f is equivalent to: f=(i=i*f)

Increment and Decrement Operators

Incrementing(increasing by 1) and decrementing(decreasing by 1). ++n increments the value of n and returns the incremented value; --n decrements the value of n and returns the decremented value. Thus the values of ++n and - -n are the new incremented or decremented values. But the value of the expression n++ is the value of n before the incrementation took place. Similar remarks apply to n--.

The if Statement

A very useful statement in the C++ language is the if statement. This statement allows to affect what happens depending on whether certain conditions are met or not.

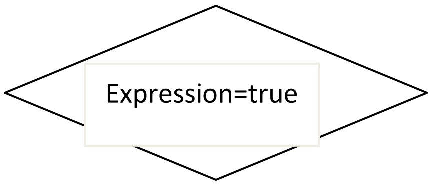
The general form of the if statement is as follows:

If(expression)

Statement1;

Statement2;

The test condition can be any expression that gives a result of true or false. If the expresssion has the value true, then statement1 is executed after which the program continues with statement2. If the expression is false then statement1 is skipped and execution continues immediately with statement2. This is illustrated in the following figure.



More than one statement to execute

```
If( expression)

{

Statement1;

Statement2;

…………;

Statement n;

}

Statement f
```

The if   else statement

Very often have to choose between two options, for example, to answer yes or no. In C++ if else statement allow a choice between two alternatives. The general form of the if statement is:

```
 If(expression)

Statement1;

else

Statement2;
```

Example

```
If (count==0)

Cout<<"the count is zero"<<endl;

else

Cout<<"the count is not equl to zero"<<endl;
```

The if and if else statements

If statement may be used in the else clause of another if statement:

```
If(choice==1)

{

......

}

else if(choice==2)

{

.......

}

else if(choice==3)

{

........

}

......

else

{

.....

}
```

In the braces, you should write any statements that should be executed when the corresponding value of the variabble choice is entered. The final else will apply to all values of the variable not covered previously.

Example

Score	Grade
80-100	A
70-79	B
60-69	C
50-59	D
40-0	F

```
If(score>=80)
Grade='A' ;
else if ( score>=70)
grade='B' ;
else if (score>=60)
grade='C' ;
else if (score>=50)
grade='D' ;
else
grade='F' ;
```

Program Control Statements

- The for Loop

The **for** is one of the most versatile statements in the C++ language because it allows a wide range of variations. The general form of the **for** loop for repeating a single statement is

for(*initialization*; *condition*; *iteration*)
statement;

For repeating a block, the general form is

for(*initialization*; *condition*; *iteration*)
{

statement sequence
}

The *initialization* is usually an assignment statement that sets the initial value of the *loop control variable,* which acts as the counter that controls the loop. The *condition* is a Boolean expression that determines whether the loop will repeat. The *iteration* expression defines the amount by which the loop control variable will change each time the loop is repeated. Notice that these three major sections of the loop must be separated by semicolons. The **for** loop will continue to execute as long as the condition tests true. Once the condition becomes false, the loop will exit, and program execution will resume on the statement following the **for**.

- **The while Loop**

The general form of the **while** loop is

while(*condition*)
{
Statement(s);
}

where *statement* can be a single statement or a block of statements, and *condition* defines the condition that controls the loop and may be any valid Boolean expression. The statement is performed while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop. The **while** is best used when the loop will repeat an unknown number of times.

- The do-while Loop

The next loop is the **do**-**while**. Unlike the **for** and the **while** loops, in which the condition is tested at the top of the loop, the **do-while** loop checks its condition at the bottom of the loop. This means that a **do-while** loop will always execute at least once. The general form of the **do while** loop is

do
{
***statements*;**
} while(*condition*);

Although the braces are not necessary when only one statement is present, they are often used to improve readability of the **do-while** construct, thus preventing confusion with the **while**.
The **do-while** loop executes as long as the conditional expression is true.

The switch statement

In C++ there is a special selection statement for else if, called switch. 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. Its objective is to check several possible constant values for an expression.   Its form is the following:

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

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

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

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

switch example

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

if-else equivalent

```
if (x == 1) {
  cout << "x is 1";
  }
else if (x == 2) {
  cout << "x is 2";
  }
else {
  cout << "value of x unknown";
  }
```

Simple Array

An array is a collection of elements of the same data type are reference by a common name. Each element of an array can be referred to by an array name and a subscript or index. Arrays can be one-dimentional or two-dimentional.

Declaring One-dimentional Arrays

The general form used to declare a single-dimensional array is:

type name[size];

The array type can be any available data type, such as int, float, char, or string. Array have indexes (subscripts) that range from 0 through one less than the number of elements in the array. Thus if an array has 5 elements, the subscripts range from 0 through 4 as illustrated in Figure 1.

1	--> A[0]
2	--> A[1]
7	--> A[2]
9	--> A[3]
13	--> A[4]

Figure 1 illustrated the storage of elements in the array

Array initialization

Initialize an array when it is declared by listing the initial values for its elements. The expanded syntax form is the following:

Type name [size]={initialization list};

The initialization list is a list of appropriate values separated by commas. For example,

int A[5]= {1,5,7,20,6}
If the number of initial values given is less than the number of elements in the array, the remaining elements are initialized to zero. Thus int A[5]={1,5} is equivalent to
int A[5]={1,5,0,0,0}.

The following program illustrates the usage of an array:

```
//This program performs input-output operations on arrays
#include<iostream.h>
void main()
{
int A[10]; //Array of size 10 is defined
int i,j;
cout<<"Input the elements of the array:"<<endl;
for(i=0; i<10; i++) //As last subscript is 9
{
cin>>A[i]; //i is called the array subscript
}
cout<<"The elements stored in the array are:"<<endl;
for(j=0; j<10; j++)
cout<<A[j]    //Displays the elements of the array
}
```

Program illustrates the input and output operations using an array. Array A is declared to be of type int and size 10. A maximum of 10 numbers can be stored in the array and the individual elements are referenced using the array subscript.

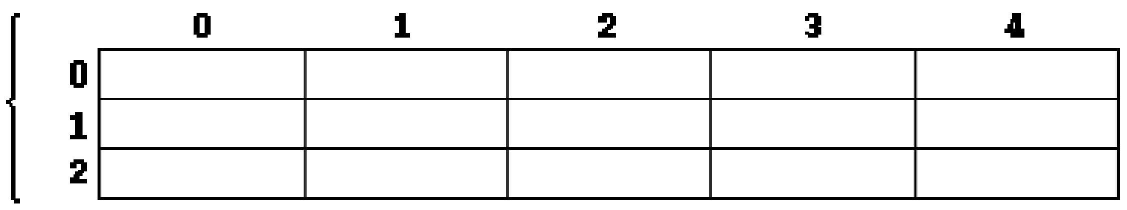
The sample output of the Program is:

```
Input the elements of the array:
1
2
7
9
13
16
3
8
100
21

The elements stored in the array are:
1 2 7 9 13 16 3 8 100 21
```

Multidimensional arrays

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



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

```
int matrix  [3][5];
```

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

```
matrix [1][3]
```

Functions

A Function which can also be referred to as subroutine, procedure, or subprogram. It groups a number of program statements into a single unit and gives it a name. A function is usually designed to perform a specific task, and its name often reflects that task. It takes in input, does something with it, and spits out an answer. Functions must have a definition and should have a declaration, although a definition can serve as a declaration if the declaration appears before the function is called.

Advantages of using functions in C++

- A complex program can be divided into small subtasks and function sub-programs can be written for each;
- These are easy to write, understand and debug;
- Write code once, and call it from many places. This insures consistency, and reduces the cost of writing and debugging.
- Functions can be collected in libraries and reused in other programs.

The general form of the function is: -

type name (parameter1, parameter2, ...) $\longrightarrow$ Function Definition

```
{

statements / function's body


}
```

where:

- type is the data type specifier of the data returned by the function.
- name is the identifier by which it will be possible to call the function. The name of the function should begin with the alphabet or underscore.

- The parameter list consists of variables separated with comma along with their data types. The parameter list could be empty which means the function do not contain any parameters. The parameter list should contain both data type and name of the variable. For example,   int factorial(int n, float j)

- `statements` is the function's body. It is a block of statements surrounded by braces `{ }`. The body of the function performs the computations.

Function Declaration

A function declaration is made by declaring the return type of the function, name of the function and the data types of the parameters of the function. A function declaration is same as the declaration of the variable. The function declaration is always terminated by the semicolon. A call to the function cannot be made unless it is declared. The general form of the declaration is:-

return_type function_name(parameter list);

For example function declaration can be: int factorial(int n1,float j1);

The variables name need not be same as the variables of parameter list of the function. Another method can be :   int factorial(int , float);

The variables in the function declaration can be optional but data types are necessary.

Function Arguments

The information is transferred to the function by the means of arguments when a call to a function is made. Arguments contain the actual value which is to be passed to the function when it is called. The sequence of the arguments in the call of the function should be same as the sequence of the parameters in the parameter list of the declaration of the function. The data types of the arguments should correspond with the data types of the parameters. When a function call is made arguments replace the parameters of the function.

The Return Statement and Return values

A return statement is used to exit from the function where it is. It returns the execution of the program to the point where the function call was made. It returns a value to the calling code. The general form of the return statement is:-   return (expression);

The expression evaluates to a value which has type same as the return type specified in the function declaration. For example the statement,   return(n);

is the return statement of the factorial function. The type of variable n should be integer as specified in the declaration of the factorial function. If a function has return type as void then return statement does not contain any expression. It is written as:-   return;

The function with return type as void can ignore the return statement. The closing braces at the end indicate the exit of the function.

Here is an example that illustrates the working of functions.

```
 #include<iostream>

int factorial(int n);        →  Prototype declaration

int main ()

{

    int n1,fact;

    cout <<"Enter the number whose factorial has to be calculated" <<  endl;

    cin >> n1;

    fact=factorial(n1);      →  Function call

    cout << "The factorial of " << n1 << "  is : " << fact << endl;

    return(0);

   }

int factorial(int n)         →  Function Definition

{

    int i, fact=1;                 ⎫
                                   ⎪
    for(i=1;i<=n;i++)              ⎪
                                   ⎪
        {                          ⎪
                                   ⎬  Body of the function
            fact=fact*i;           ⎪
                                   ⎪
        }                          ⎪
                                   ⎪
        return(fact);              ⎭

 }
```

Multiple function

Each of functions will have a prototype after the header and each have its own function definition after main program. For example, design a main program that uses 2 different functions. Function that can multiply 2 numbers, and another that can multiply 4 numbers.

```
#include <iostream.h>
#include <conio.h >
int MultTwo(int x,int y);                          //prototypes
int MultFour(int m,int n,int o,int p);             //prototypes

int a,b,c,d,e;

int main()
{
a=1;
b=2;
c=MultTwo(a,b);        //function call
d=5;
e=MultFour(a,b,c,d);   //function call
cout << e << endl;

return 0;
}
int MultTwo(int x, int y)
{
return (x*y);
}
int MultFour(int m, int n, int o, int p)
{
return (m*n*o*p);

}
```

Void Functions

Void functions are created and used just like value-returning functions except they do not return a value after the function executes. A void function just "does something" and then returns.

```
#include <iostream.h>

void Add(int x, int y);
```

```
int a,b;

int main()
{
a=1;
b=2;
Add(a,b);

return 0;
}

void Add(int x, int y)
{
int z;
z=(x+y);
cout << z << endl;
}
```