المادة الدراسية :- البرمجة بلغة ++ C

مدرس المادة :- م.م. دريد ثامر سالم

المرحلة :- الثانية

الكورس الدراسي :- الاول

Introduction

A programming language is a language that can be used to write computer programs which control functionality or behaviour of a computer. In simple language it controls the way in which a computer is operated or work. A programming language is not a spoken language. It is a way of describing what the programmer wants the computer to do. In fact, every programming language is defined by the syntactic and semantic rules which describe the whole language.

Suppose you landed into trouble and to solve it there are various ways or steps. The steps followed by you in real life to solve a problem is similar to the instruction given by you to solve a particular problem in a programming language. That is instead of following it you need to code it. The steps are provided by you as instructions to the computer as programs written is a particular language.

PROGRAM: A computer program is also called piece of code or source code and the actual writing of source code is called coding.

C programs are written in high-level language using letters, numbers, and other symbols that you can easily find on computer keyboard.

HIGH LEVEL LANGUAGE: A high level language is human understandable language. It is written using all the keys in the keyboard but in reality computer understand a low level language.

Computersactually execute low-level machine language (also known as binary number).

LOW LEVEL MACHINE LANGUAGE: A low-level machine language is computer understandable language. It is also called binary number. A binary number contains only zero(0) and one(1).

The C is a general-purpose, procedural, imperative computer programming language developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX operating system.

The C is the most widely used computer language, it keeps fluctuating at number one scale of popularity along with Java programming language, which is also equally popular and most widely used among modern software programmers

CHARACTER SET

A character set defines the valid character that can be used in a source program. A character set is the smallest unit that provides meaningful information to compiler. Characters are combined to form varible names, defining data typed, constants, statements, and programs in C.

The character set used in C language is divided into the following categories

1. **Alphabets**

* Uppercase letters: A,B,C,...,Z

* Lowercase letters: a,b,c,....,z

2. **Digits**

* 0,1,2,3,4,5,6,7,8,9

3. **Special Characters**

* ~ ! @ # $ % ^ & * ( ) [] {} <> \ / etc.

4. **White Space Character**

* Carriage return

* New line character

* form feed character

* Backspace character

* Horizontal tab space character

component

Now lets begin exploring the C programming language. Before we start executing programs in C Turbo C/C++ Compiler we must know some basic component of a C program.

#include - The #include is known as a preprocessor directive and is used to tell the C preprocessor to find the stdio file with extension .h. <stdio.h> stand for standard input output stream header file and contains information for printf, scanf etc.

main() - Exectuion of a a program starts from a main() function. It defines the point from where the execution of the program starts. Anything written between opening curly brace and ending curly brace of main is executed.

printf() - This is the standard way of producing output. The functionality of printf() is referenced in stdio.h by the C compiler, thus it always work in the same way.

scanf() - This is the standard way of taking input from user. The functionality of scanf() is also referenced in stdio.h by the C compiler.

comments: Comments are information given by the program to make a program readable and easy to understand. It reduces the complexity of a program. Anything written as comments is ignored by the compiler.

There are two ways of writing comments:

1. Single line comment

Syntax://Your comments here

example:

// Hello this is my first C program

2. Multiline comment

Syntax:

/* Your comment

Your comment*/

example:

/* This is my first C program

and i am very excited about it */

SYNTAX:

Note : It should be noted that some compiler does not include header file "conio.h". It is also not included in compiler of Linux or any other Unix based operating system. So in case using it displays an error remove this and the function associated with it i.e getch().

```
#include<stdio.h> //This tells the compiler about the input/output functions such as printf(),
scanf()

#include<conio.h> //It is used for getch() function

int  main() // It is the entry point of a program

{ // Progam begins with this curly braces

printf("Congratulation you successfully run your  first program"); // to print in the output screen

getch(); // to hold the output screen
return 0; // tell the OS that the program exited without error

} // Program end with this curly braces
```

OUTPUT:

Congratulation you successfully run your first program

EXPLANATION:

return 0 means that the program is terminated successfully and the compiler is returning back the control to the computer

Now on execution the program will display "Congratulation you successfully run your first program" (without double quotes) as output on monitor. Any string you pass within double quotes through *printf* is sent to console output (monitor) i.e it is displayed as it is on the monitor.

PROGRAM EXAMPLE:

```
#include<stdio.h>

int main()

{

printf("%d", 5+6); // %d is conversion character for integer

return 0;

}
```

OUTPUT:

11

EXPLANATION:

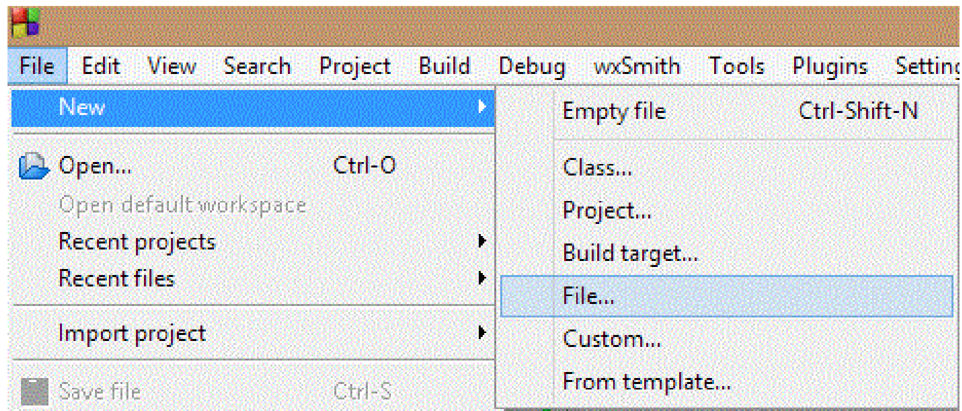
The above program contains only one statement to be executed within main() function. As this statement contains 5+6 without quotes, it will add these two numbers and pass the result in integer (as indicated by conversion character %d) to console window i.e 11 is sent to the monitor. So in this case the output is 11.

So whenever we use double quotes, compiler just prints the message ignoring what has been include there but when no quotes are used, actual values are processed according to instructions and executed accordingly.

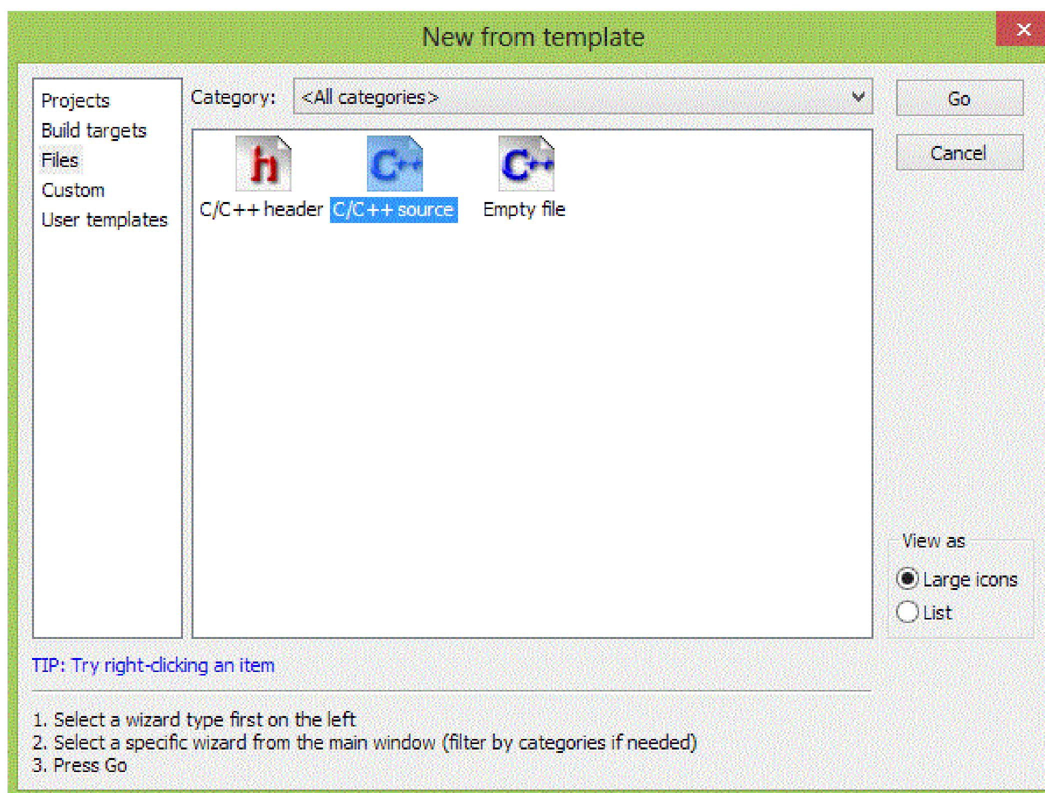
How to create program in c language

Follow these steps to write your first C Program:

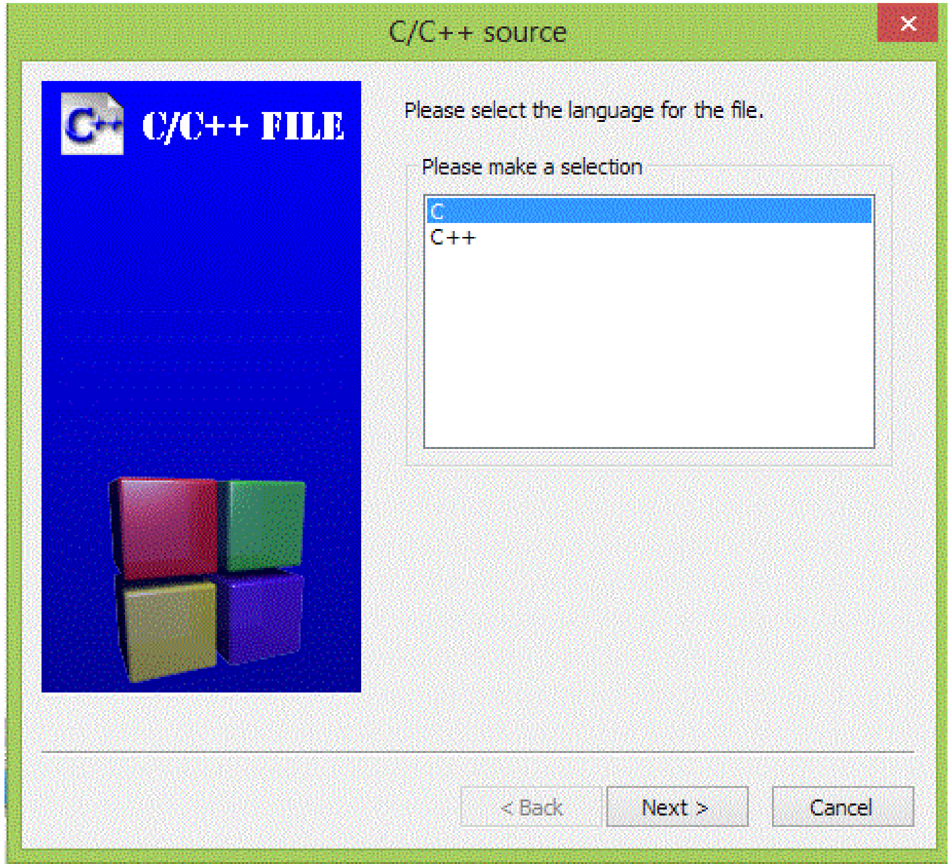
- **Click on File - New - File...**



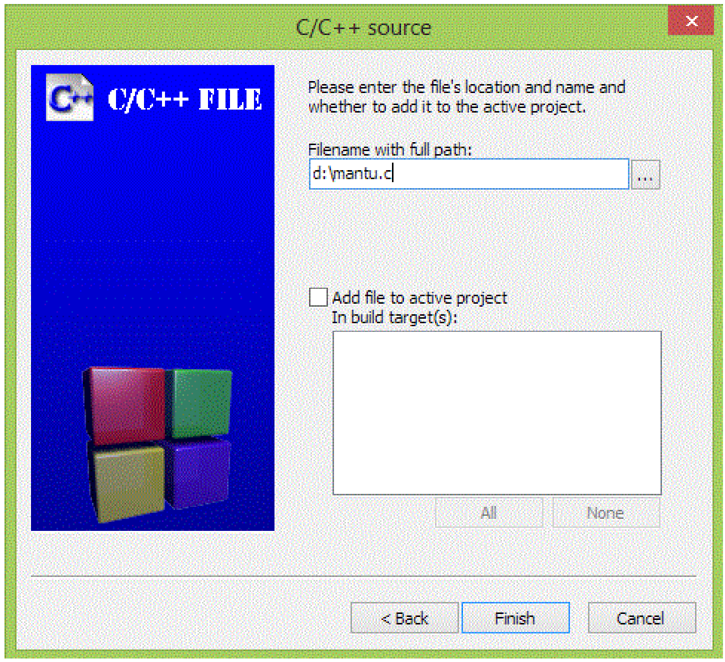
Select C/C++ Source and click on Go.



A window for Language Selection pops up. Select C and click on Next.



Enter the File Location with File Name i.e where you want to save your file and click on Finish.



HOW TO COMPILE AND RUN C PROGRAM

- To Compile c program : Click on Build - Compile Current File. You can also use shortcut key - Ctrl + Shift + F9
- To Run c program : Click on Build - Run. You can also use shortcut key - Ctrl + F10
- To Compile and run c program at the same time : Click on Build - Build and Run. you can achieve the same task by pressing F9

Lets look at how to save the source code in a file, and how to compile and run it. Following are the simple steps:

1. Open a text editor and add the above-mentioned code.
2. Save the file as *hello.c*
3. Open a command prompt and go to the directory where you saved the file.
4. Type *gcc hello.c* and press enter to compile your code.
5. If there are no errors in your code the command prompt will take you to the next line and would generate *a.out* executable file.
6. Now, type *a.out* to execute your program.
7. You will be able to see *"Hello World"* printed on the screen

```
$ gcc hello.c
$ ./a.out
Hello, World!
```

Make sure that gcc compiler is in your path and that you are running it in the directory containing source file hello.c.

conio.h is a C header file used mostly by MS-DOS compilers to provide console input/output. It is not part of the C standard library or ISO C, nor is it defined by POSIX.

This header declares several useful library functions for performing "console input and output" from a program.

Variables :-

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Syntax to declare a variable:-

Variable type _variable name;

The example of declaring variable is given below:

int a;

float b;

char c;

Here, a, b, c are variables and int,float,char are data types.

We can also provide values while declaring the variables as given below:

int a=10,b=20;//declaring 2 variable of integer type

float f=20.8;

char c='A';

There are many types of variables in c:

1. local variable
2. global variable
3. static variable
4. automatic variable
5. external variable

Local Variable

A variable that is declared inside the function or block is called local variable.

It must be declared at the start of the block.

```
void function1(){
int x=10;//local variable
}
```

You must have to initialize the local variable before it is used.

Global Variable

A variable that is declared outside the function or block is called global variable. Any function can change the value of the global variable. It is available to all the functions.

It must be declared at the start of the block.

```
int value=20;//global variable
void function1(){
int x=10;//local variable
}
```

Static Variable

A variable that is declared with static keyword is called static variable.

It retains its value between multiple function calls.

```
void function1(){
int x=10;//local variable
static int y=10;//static variable
x=x+1;
```

```
y=y+1;

printf("%d,%d",x,y);

}
```

If you call this function many times, **local variable will print the same value** for each function call e.g, 11,11,11 and so on. But **static variable will print the incremented value** in each function call e.g. 11, 12, 13 and so on.

Automatic Variable

All variables in C that is declared inside the block, are automatic variables by default. By we can explicitly declare automatic variable using **auto keyword**.

```
void main(){

int x=10;//local variable (also automatic)

auto int y=20;//automatic variable

}
```

External Variable

We can share a variable in multiple C source files by using external variable. To declare a external variable, you need to use **extern keyword**.

myfile.h

```
extern int x=10;//external variable (also global)
```

program1.c

```
nput#include "myfile.h"

#include <stdio.h>

void printValue(){

printf("Global variable: %d", global_variable);
```

```
}
```

Input numbers from the keyboard :-

The printf() and scanf() functions are used for input and output in C language. Both functions are inbuilt library functions, defined in stdio.h (header file).

printf() function

The **printf() function** is used for output. It prints the given statement to the console.

The syntax of printf() function is given below:

```
printf("format string",argument_list);
```

The **format string** can be %d (integer), %c (character), %s (string), %f (float) etc.

scanf() function

The **scanf() function** is used for input. It reads the input data from the console.

```
scanf("format string",argument_list);
```

Program to print cube of given number

Let's see a simple example of c language that gets input from the user and prints the cube of the given number.

```
#include<stdio.h>

#include<conio.h>

void main(){

int number;

clrscr();

printf("enter a number:");

scanf("%d",&number);
```

```
printf("cube of number is:%d ",number*number*number);

getch();

}
```

Output

enter a number:5

cube of number is:125

The **scanf("%d",&number)** statement reads integer number from the console and stores the given value in number variable.

The **printf("cube of number is:%d ",number*number*number)** statement prints the cube of number on the console.

Program to print sum of 2 numbers

Let's see a simple example of input and output in C language that prints addition of 2 numbers.

```
#include<stdio.h>

#include<conio.h>

void main(){

int x=0,y=0,result=0;

clrscr();

printf("enter first number:");

scanf("%d",&x);

printf("enter second number:");

scanf("%d",&y);
```

```
result=x+y;
printf("sum of 2 numbers:%d ",result);
getch();
}
```

Output

enter first number:9

enter second number:9

sum of 2 numbers:18

Sample program illustrating use of scanf() to read integers, characters and floats

```
#include < stdio.h >
main()
{
    int sum;
    char letter;
    float money;
    printf("Please enter an integer value ");
    scanf("%d", &sum );
    printf("Please enter a character ");
    /* the leading space before the %c ignores space characters in the input */
    scanf("  %c", &letter );
    printf("Please enter a float variable ");
    scanf("%f", &money );
```

```
    printf("\nThe variables you entered were\n");
    printf("value of sum = %d\n", sum );
    printf("value of letter = %c\n", letter );
    printf("value of money = %f\n", money );
}
```

Sample Program Output

Please enter an integer value

34

Please enter a character

W

Please enter a float variable

32.3

The variables you entered were

value of sum = 34

value of letter = W

value of money = 32.300000

functions in C

The **function in C language** is also known as *procedure* or *subroutine* in other programming languages.

To perform any task, we can create function. A function can be called many times. It provides *modularity* and code *reusability*.

Advantage of functions in C

There are many advantages of functions.

1) Code Reusability

By creating functions in C, you can call it many times. So we don't need to write the same code again and again.

2) Code optimization

It makes the code optimized, we don't need to write much code.

Suppose, you have to check 3 numbers (781, 883 and 531) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

But if you use functions, you need to write the logic only once and you can reuse it several times.

Syntax to declare function in C

The syntax of creating function in c language is given below:

```
return_type function_name(data_type parameter...){
//code to be executed
}
```

Syntax to call function in C

The syntax of calling function in c language is given below:

```
variable=function_name(arguments...);
```

1) variable: The variable is not mandatory. If function return type is *void*, you must not provide the variable because void functions doesn't return any value.

2) function_name: The function_name is name of the function to be called.

3) arguments: You need to provide same number of arguments as defined in the function at the time of declaration or definition.

Example of function in C

Let's see the simple program of function in c language.

```
#include <stdio.h>
#include <conio.h>
//defining function
int cube(int n){
return n*n*n;
}
void main(){
int result1=0,result2=0;
clrscr();
result1=cube(2);//calling function
result2=cube(3);
printf("%d \n",result1);
printf("%d \n",result2);
getch();
}
```

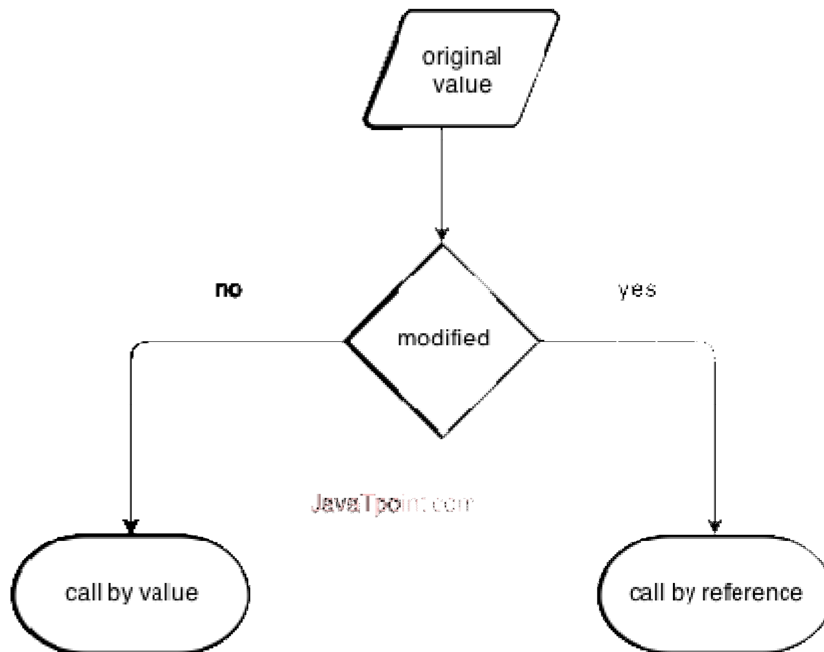
Output

8

27

Call by value and call by reference in C

There are two ways to pass value or data to function in C language: *call by value* and *call by reference*. Original value is not modified in call by value but it is modified in call by reference.



Call by value in C

In call by value, **original value is not modified**.

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Example :-

```
#include <stdio.h>

#include <conio.h>

void change(int num) {

printf("Before adding value inside function num=%d \n",num);
```

```
num=num+100;
 printf("After adding value inside function num=%d \n", num);
}
int main() {
int x=100;
clrscr();
printf("Before function call x=%d \n", x);
change(x);//passing value in function
printf("After function call x=%d \n", x);
getch();
return 0;
}
```

Output

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=100

Call by reference in C

In call by reference, **original value is modified** because we pass reference (address).

Here, address of the value is passed in the function, so actual and formal arguments shares the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

Note: To understand the call by reference, you must have the basic knowledge of pointers.

Example :-

```
#include <stdio.h>

#include <conio.h>

void change(int *num) {

printf("Before adding value inside function num=%d \n",*num);

 (*num) += 100;

 printf("After adding value inside function num=%d \n", *num);

}

int main() {

int x=100;

clrscr();

printf("Before function call x=%d \n", x);

change(&x);//passing reference in function

printf("After function call x=%d \n", x);

getch();

return 0;

}
```

Output

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=200

Difference between call by value and call by reference in c

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

Recursion in C

When *function is called within the same function*, it is known as **recursion** in C. The function which calls the same function, is known as **recursive function**.

A function that calls itself, and doesn't perform any task after function call, is know as **tail recursion**. In tail recursion, we generally call the same function with return statement. An example of tail recursion is given below.

example :-

```
recursionfunction(){

recursionfunction();//calling self function

}
```

Example of tail recursion in C

```
#include<stdio.h>

#include<conio.h>
```

```
int factorial (int n)
{
if ( n < 0)
return -1; /*Wrong value*/
if (n == 0)
return 1; /*Terminating condition*/
return (n * factorial (n -1));
}
void main(){
int fact=0;
clrscr();
fact=factorial(5);
printf("\n factorial of 5 is %d",fact);
getch();
}
```

Output

factorial of 5 is 120

```
return 5 * factorial(4)   120
    return 4 * factorial(3) – 24
        return 3 * factorial(2) – 6
            return 2 * factorial(1)   2
                return 1 * factorial(0) – 1

1 * 2 * 3 * 4 * 5 = 120
```

Fig: Recursion

Introduction in C++

C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming.

C++ is regarded as a **middle-level** language, as it comprises a combination of both high-level and low-level language features.

C++ was developed by Bjarne Stroustrup starting in 1979 at Bell Labs in Murray Hill, New Jersey, as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.

C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

Note: A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time.

one after another without getting repeated or ignored. Certain tasks require execution

C++ Basic Input/Output

The C++ standard libraries provide an extensive set of input/output capabilities which we will see in subsequent chapters. This chapter will discuss very basic and most common I/O operations required for C++ programming.

C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc, this is called **output operation**.

C++ Program Structure:

Let us look at a simple code that would print the words **Hello World.**

```
#include <iostream>
using namespace std;
// main() is where program execution begins.
int main()
{
```

```
    cout << "Hello World"; // prints Hello World
    return 0;
}
```

Let us look various parts of the above program:

- The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header **<iostream>** is needed.
- The line **using namespace std;** tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++.
- The next line **// main() is where program execution begins.** is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.
- The line **int main()** is the main function where program execution begins.
- The next line **cout << "This is my first C++ program.";** causes the message "This is my first C++ program" to be displayed on the screen.
- The next line **return 0;** terminates main( )function and causes it to return the value 0 to the calling process

I/O Library Header Files:

There are following header files important to C++ programs:

Header File	Function and Description
<iostream>	This file defines the **cin, cout, cerr** and **clog** objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
<iomanip>	This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as **setw** and **setprecision**.
<fstream>	This file declares services for user-controlled file processing.

The standard output stream (cout):

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

```
#include <iostream>
using namespace std;
int main( )
{
   char str[] = "Hello C++";
   cout << "Value of str is : " << str << endl;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of str is : Hello C++
```

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.

The insertion operator << may be used more than once in a single statement as shown above and **endl** is used to add a new-line at the end of the line.

The standard input stream (cin):

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

```
#include <iostream>
using namespace std;
int main( )
{
```

```
    char name[50];
    cout << "Please enter your name: ";
    cin >> name;
    cout << "Your name is: " << name << endl;
}
```

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the result something as follows:

```
Please enter your name: cplusplus
Your name is: cplusplus
```

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.

The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following:

```
cin >> name >> age;
```

This will be equivalent to the following two statements:

```
cin >> name;
cin >> age;
```

The standard error stream (cerr):

The predefined object **cerr** is an instance of **ostream** class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object **cerr** is un-buffered and each stream insertion to cerr causes its output to appear immediately.

The **cerr** is also used in conjunction with the stream insertion operator as shown in the following example.

```
#include <iostream>
using namespace std;
int main( )
{
```

```
   char str[] = "Unable to read....";
   cerr << "Error message : " << str << endl;
}
```

When the above code is compiled and executed, it produces the following result:

```
Error message : Unable to read....
```

The standard log stream (clog):

The predefined object **clog** is an instance of **ostream** class. The clog object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered. This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed.

The **clog** is also used in conjunction with the stream insertion operator as shown in the following example.

```
#include <iostream>
using namespace std;
int main( )
{
   char str[] = "Unable to read....";
   clog << "Error message : " << str << endl;
}
```

When the above code is compiled and executed, it produces the following result:

```
Error message : Unable to read....
```

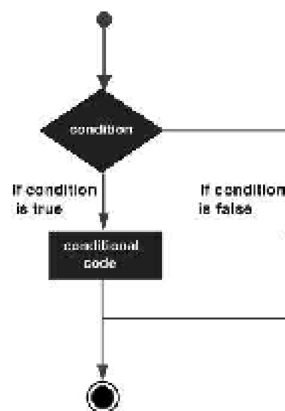
You would not be able to see any difference in cout, cerr and clog with these small examples, but while writing and executing big programs then difference becomes obvious. So this is good practice to display error messages using cerr stream and while displaying other log messages then clog should be used.

Control Statements in C++

Program that we tried so far are executed in an orderly manner i.e. the statements are executed of some statements ignoring the rest. These can be accomplishing by using control statements.

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general from of a typical decision making structure found in most of the programming languages:



C++ programming language provides following types of control statements.

1. **if statement :-**

An **if** statement consists of a Boolean expression followed by one or more statements.

Syntax:

The syntax of an if statement in C++ is:

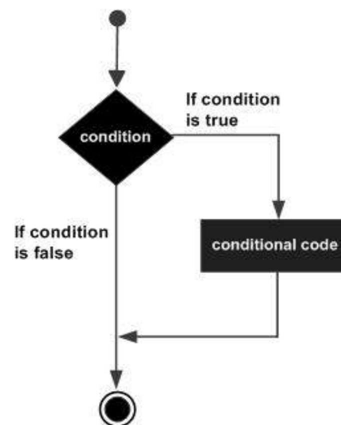
if(boolean_expression)

{

```
   // statement(s) will execute if the boolean expression is true
}
```

If the boolean expression evaluates to **true**, then the block of code inside the if statement will be executed. If boolean expression evaluates to **false**, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Flow Diagram:



Example:

```
#include <iostream>
using namespace std;
int main ()
{
   // local variable declaration:
   int a = 10;
   // check the boolean condition
   if( a < 20 )
   {
       // if condition is true then print the following
       cout << "a is less than 20;" << endl;
```

```
    }
    cout << "value of a is : " << a << endl;
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
a is less than 20;
value of a is : 10
```

2. **if...else statement :-**

An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

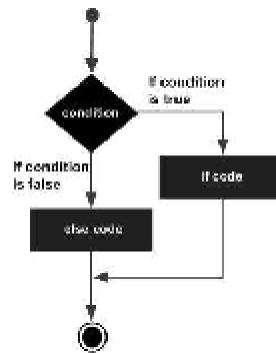
Syntax:

The syntax of an if...else statement is :-

```
if(boolean_expression)
{
  // statement(s) will execute if the boolean expression is true
}
else
{
  // statement(s) will execute if the boolean expression is false
}
```

If the boolean expression evaluates to **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.

Flow Diagram:



Example:

```
#include <iostream>

using namespace std;

int main ()

{

   // local variable declaration:

   int a = 100;

   // check the boolean condition

   if( a < 20 )

   {

       // if condition is true then print the following

       cout << "a is less than 20;" << endl;

   }

   else

   {

       // if condition is false then print the following

       cout << "a is not less than 20;" << endl;

   }
```

```
   cout << "value of a is : " << a << endl;

   return 0;

}
```

When the above code is compiled and executed, it produces the following result:

```
a is not less than 20;

value of a is : 100
```

3. The if...else if...else Statement:

An **if** statement can be followed by an optional **else if...else** statement, which is very usefull to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of he remaining else if's or else's will be tested.

Syntax:

The syntax of an if...else if...else statement in C++ is:

```
if(boolean_expression 1)

{

   // Executes when the boolean expression 1 is true

}

else if( boolean_expression 2)

{

   // Executes when the boolean expression 2 is true

}

else if( boolean_expression 3)

{

   // Executes when the boolean expression 3 is true
```

```
}
else
{
  // executes when the none of the above condition is true.
}
```

Example:

```
#include <iostream>
using namespace std;
int main ()
{
  // local variable declaration:
  int a = 100;
  // check the boolean condition
  if( a == 10 )
  {
     // if condition is true then print the following
     cout << "Value of a is 10" << endl;
  }
  else if( a == 20 )
  {
     // if else if condition is true
     cout << "Value of a is 20" << endl;
  }
  else if( a == 30 )
  {
     // if else if condition is true
```

```
    cout << "Value of a is 30" << endl;
  }
  else
  {
    // if none of the conditions is true
    cout << "Value of a is not matching" << endl;
  }
  cout << "Exact value of a is : " << a << endl;
  return 0;}
```

When the above code is compiled and executed, it produces the following result:

Value of a is not matching , Exact value of a is : 100

4- Nested if statements :

It is always legal to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax:

The syntax for a **nested if** statement is as follows:

```
if( boolean_expression 1)
{
  // Executes when the boolean expression 1 is true
  if(boolean_expression 2)
  {
    // Executes when the boolean expression 2 is true
  }
}
```

You can nest **else if...else** in the similar way as you have nested *if* statement.

Example:

```
#include <iostream>
using namespace std;
int main ()
{
  // local variable declaration:
  int a = 100;
  int b = 200;
  // check the boolean condition
  if( a == 100 )
  {
     // if condition is true then check the following
     if( b == 200 )
     {
        // if condition is true then print the following
        cout << "Value of a is 100 and b is 200" << endl;
     }
  }
  cout << "Exact value of a is : " << a << endl;
  cout << "Exact value of b is : " << b << endl;
  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

5- switch statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

The syntax for a **switch** statement in C++ is as follows:

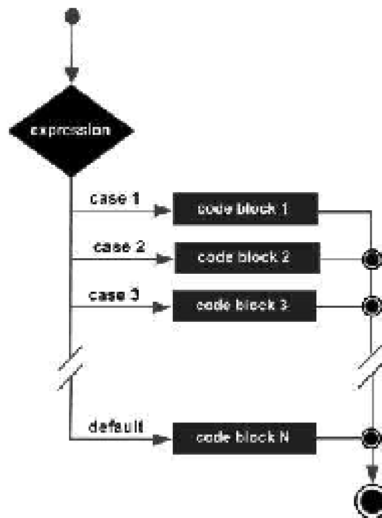
```
switch(expression){
   case constant-expression  :
      statement(s);
      break; //optional
   case constant-expression  :
      statement(s);
      break; //optional
   // you can have any number of case statements.
   default : //Optional
      statement(s);
}
```

The following rules apply to a switch statement:

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Flow Diagram:



Example:

```
#include <iostream>
using namespace std;
int main ()
{
   // local variable declaration:
   char grade = 'D';
   switch(grade)
   {
```

```
  case 'A' :
    cout << "Excellent!" << endl;
    break;
  case 'B' :
  case 'C' :
    cout << "Well done" << endl;
    break;
  case 'D' :
    cout << "You passed" << endl;
    break;
  case 'F' :
    cout << "Better try again" << endl;
    break;
  default :
    cout << "Invalid grade" << endl;
  }
  cout << "Your grade is " << grade << endl;
  return 0;
}
```

This would produce the following result:

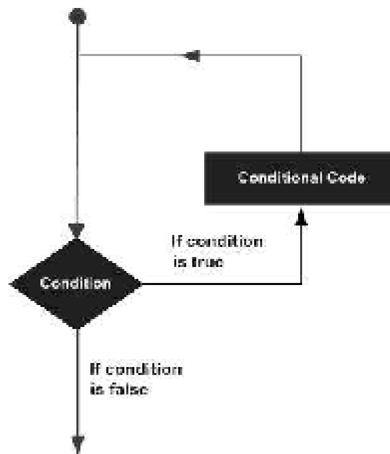
```
You passed
Your grade is D
```

Loop Types

There may be a situation, when you need to execute a block of code several number of times. In general statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:



C++ programming language provides the following types of loop to handle looping requirements.

1. for loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

The syntax of a for loop in C++ is:

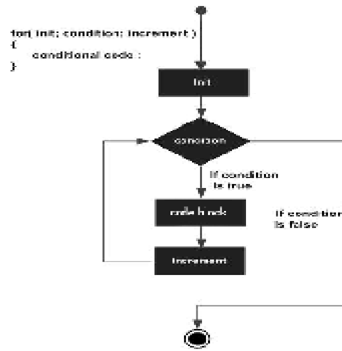
```
for ( init; condition; increment )
{
  statement(s);
}
```

Here is the flow of control in a for loop:

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram:



Example:

```
#include <iostream>
using namespace std;
int main ()
{
  // for loop execution
  for( int a = 10; a < 20; a = a + 1 )
  {
     cout << "value of a: " << a << endl;
  }
  return 0;
}
```

2. nested loops

A loop can be nested inside of another loop. C++ allows at least 256 levels of nesting.

Syntax:

The syntax for a **nested for loop** statement in C++ is as follows:

for (init; condition; increment)

{

```
  for ( init; condition; increment )
  {
    statement(s);
  }
  statement(s); // you can put more statements.
}
```

The syntax for a **nested while loop** statement in C++ is as follows:

```
while(condition)
{
  while(condition)
  {
    statement(s);
  }
  statement(s); // you can put more statements.
}
```

The syntax for a **nested do...while loop** statement in C++ is as follows:

```
do
{
  statement(s); // you can put more statements.
  do
  {
    statement(s);
  }while( condition );
}while( condition );
```

Example:

The following program uses a nested for loop to find the prime numbers from 2 to 100:

```
#include <iostream>
using namespace std;
int main ()
{
  int i, j;
  for(i=2; i<100; i++) {
    for(j=2; j <= (i/j); j++)
      if(!(i%j)) break; // if factor found, not prime
      if(j > (i/j)) cout << i << " is prime\n";
  }
  return 0;
}
```

3. do...while loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

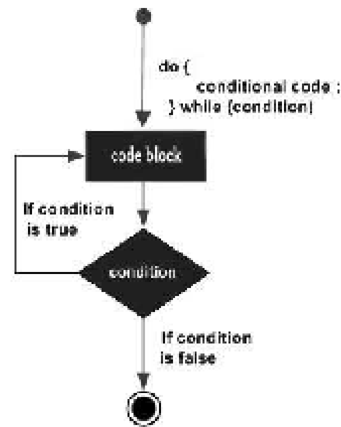
The syntax of a do...while loop in C++ is:

```
do
{
  statement(s);
}while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

Flow Diagram:



Example:

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

int main ()
{
  // Local variable declaration:
  int a = 10;
  // do loop execution
  do
  {
     cout << "value of a: " << a << endl;
     a = a + 1;
  }while( a < 20 );
  return 0;
}
```

1. break statement

The **break** statement has the following two usages in C++:

- When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.

- It can be used to terminate a case in the **switch** statement (covered in the next chapter).

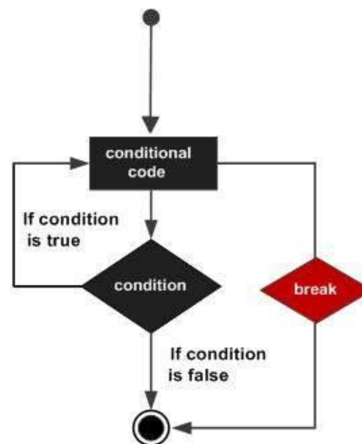
If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax of a break statement in C++ is:

```
break;
```

Flow Diagram:



Example:

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

int main ()
{
   // Local variable declaration:
   int a = 10;
   // do loop execution
```

```
  do
  {
     cout << "value of a: " << a << endl;
     a = a + 1;
     if( a > 15)
     {
        // terminate the loop
        break;
     }
  }while( a < 20 );
  return 0;
}
```

2. continue statement

The **continue** statement works somewhat like the break statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.

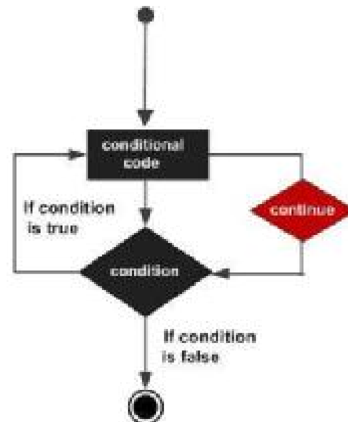
For the **for** loop, continue causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, program control passes to the conditional tests.

Syntax:

The syntax of a continue statement in C++ is:

```
continue;
```

Flow Diagram:



Example:

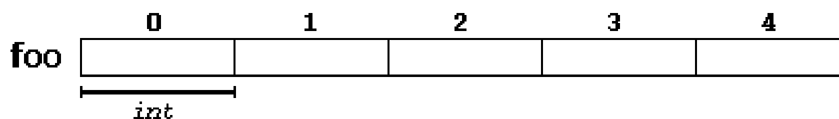
```
#include <iostream>
using namespace std;
int main ()
{
  // Local variable declaration:
  int a = 10;
  // do loop execution
  do
  {
     if( a == 15)
     {
        a = a + 1;  // skip the iteration.
        continue;
     }
     cout << "value of a: " << a << endl;
     a = a + 1;
  }while( a < 20 );
  return 0;}
```

Array in C++

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

That means that, for example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.

For example, an array containing 5 integer values of type int called foo could be represented as:



where each blank panel represents an element of the array. In this case, these are values of type int. These elements are numbered from 0 to 4, being 0 the first and 4 the last; In C++, the first element in an array is always numbered with a zero (not a one), no matter its length.

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

type name [elements];

where type is a valid type (such as int, float...), name is a valid identifier and the elements field (which is always enclosed in square brackets []), specifies the length of the array in terms of the number of elements.

Therefore, the foo array, with five elements of type int, can be declared as:

```
int foo [5];
```

NOTE: The elements field within square brackets [], representing the number of elements in the array, must be a *constant expression*, since arrays are blocks of static memory whose size must be determined at compile time, before the program runs.

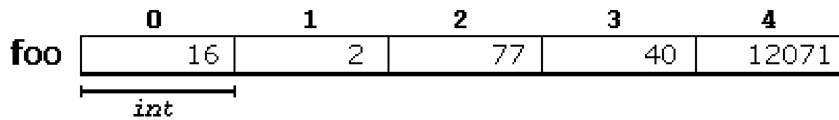
Initializing arrays

By default, regular arrays of *local scope* (for example, those declared within a function) are left uninitialized. This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared.

But the elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces {}. For example:

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

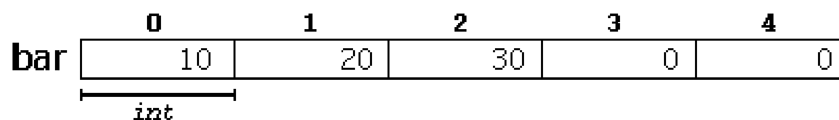
This statement declares an array that can be represented like this:



The number of values between braces {} shall not be greater than the number of elements in the array. For example, in the example above, `foo` was declared having 5 elements (as specified by the number enclosed in square brackets, []), and the braces {} contained exactly 5 values, one for each element. If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes). For example:

```
int bar [5] = { 10, 20, 30 };
```

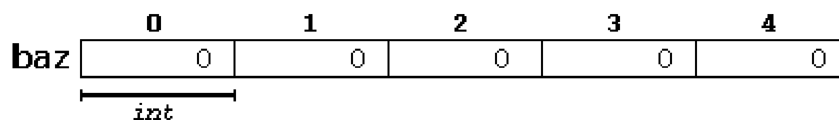
Will create an array like this:



The initializer can even have no values, just the braces:

```
int baz [5] = { };
```

This creates an array of five `int` values, each initialized with a value of zero:



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

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

After this declaration, array `foo` would be 5 `int` long, since we have provided 5 initialization values.

Finally, the evolution of C++ has led to the adoption of *universal initialization* also for arrays. Therefore, there is no longer need for the equal sign between the declaration and the initializer. Both these statements are equivalent:

```
1 int foo[] = { 10, 20, 30 };
2 int foo[] { 10, 20, 30 };
```

Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types).

Accessing the values of an array

The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:

`name[index]`
Following the previous examples in which `foo` had 5 elements and each of those elements was of type `int`, the name which can be used to refer to each element is the following:

For example, the following statement stores the value 75 in the third element of `foo`:

```
foo [2] = 75;
```

for example, the following copies the value of the third element of `foo` to a variable called `x`:

```
x = foo[2];
```

Therefore, the expression `foo[2]` is itself a variable of type `int`.

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

In C++, it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause errors on compilation, but can cause errors on runtime. The reason for this being allowed will be seen in a later chapter when pointers are introduced.

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

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

The main difference is that the declaration is preceded by the type of the elements, while the access is not.

Some other valid operations with arrays:

```
1 foo[0] = a;
2 foo[a] = 75;
3 b = foo [a+2];
4 foo[foo[a]] = foo[2] + 5;
```

For example:

```
// arrays example
#include <iostream>
```

```
usingnamespace std;

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

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

Two-Dimensional Arrays:

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows:

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C++ data type and **arrayName** will be a valid C++ identifier.

A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form **a[i][j]**, where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays:

Multidimensioned arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row have 4 columns.

```
int a[3][4] = {
 {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
```

```
{4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
{8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements:

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above digram.

```
#include <iostream>
using namespace std;
int main ()
{
  // an array with 5 rows and 2 columns.
  int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
  // output each array element's value
  for ( int i = 0; i < 5; i++ )
    for ( int j = 0; j < 2; j++ )
    {
      cout << "a[" << i << "][" << j << "]: ";
      cout << a[i][j]<< endl;
}
  return 0;
}
```