



Basic Concepts of C++

Sisoft Technologies Pvt Ltd
SRC E7, Shipra Riviera Bazar, Gyan Khand-3, Indirapuram, Ghaziabad
Website: www.sisoft.in Email: info@sisoft.in
Phone: +91-9999-283-283



Sample Program of C ++

```
/*  
    This is a sample of c ++ Program.  
*/  
# include <iostream.h>  
Using namespace std;  
  
Int main()  
{  
Cout<<"Hello everybody";           // cout for print  
Return 0;  
}
```

Output: Hello Everybody



Comments in C ++



Comment in C++ Programming is similar as that of in C. They are non Executable Statements, and used for documentation. Comments can be written anywhere and any number of times.

Comments are always neglected by compiler. They replaced by white spaces during the preprocessor phase.

Types of comments:

1. Single Comments :

Single line comments starts with “//” Symbol.

- Ex: `Cout<<“Hello World” // Print Hello World`
- `Cout<<“Good Morning” // Print Good Morning`

2. Multiple line Comments:

Multiple line comments starts with `/*` and end with `*/`

Ex: `/* this is`

- `A sample program of c ++`
- `*/`



Cin keyword



Cin : Extracting Input from User Using Keyboard

- In C++, Cin is used for accepting data from keyboard.
- The operator >> called as extraction operator or get from operator. This is used to accept value from the user.
- >> can be overloaded.
- Cin is the object of istream class

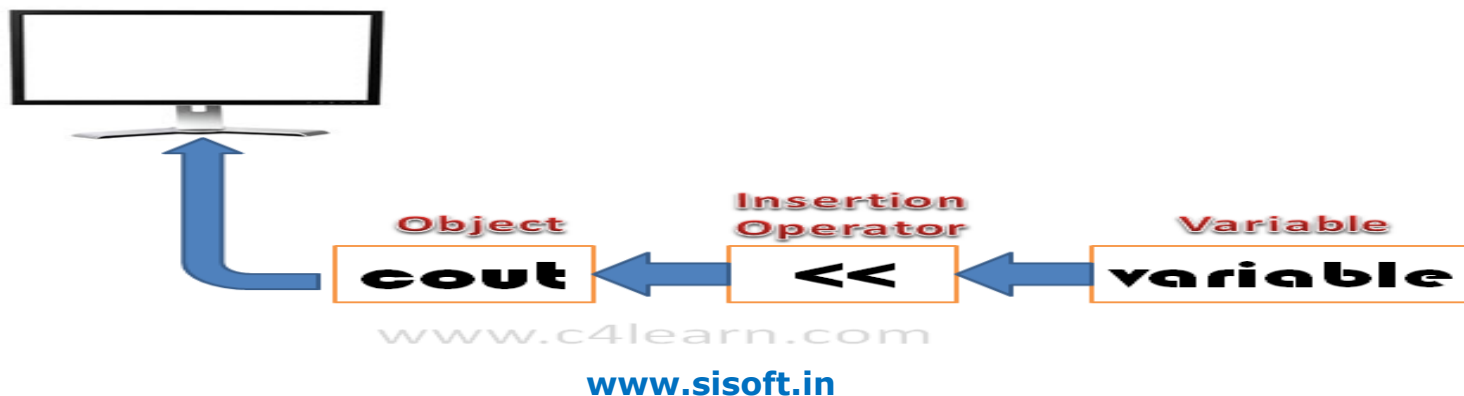
- Syntax: `cin>>variable`



Cout keyword

Cout : Display Output to User Using Screen(Monitor)

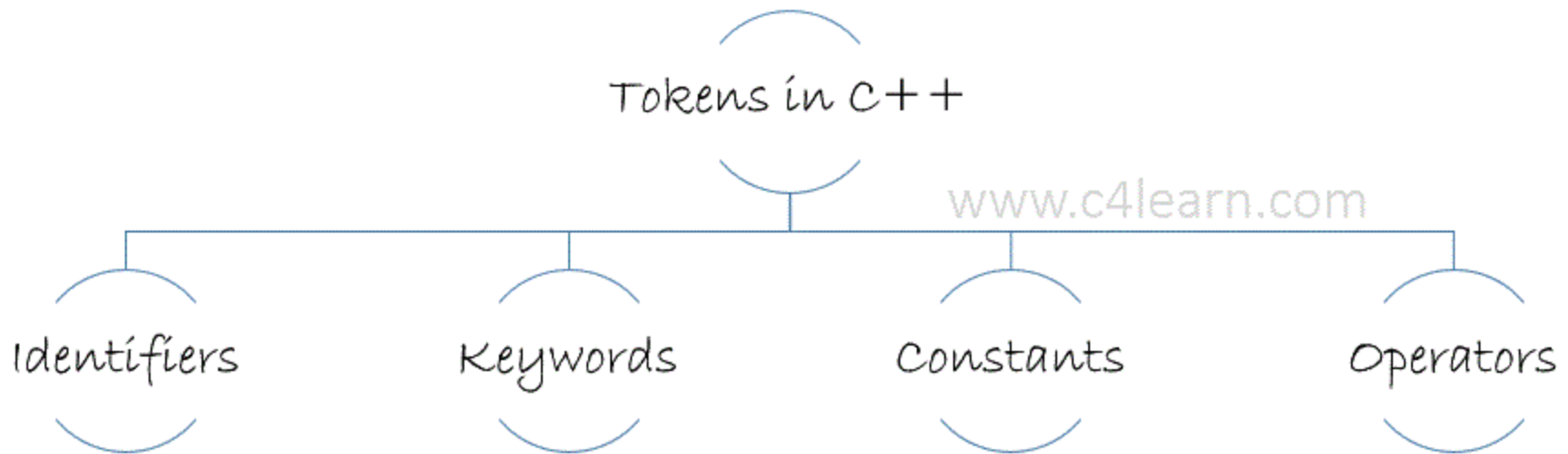
- In C++ , Cout is used for displaying data on the screen.
- The operator << called as insertion operator or put to operator. It is used to display value to the user. The value may be some message in the form of string or variable.
- << can be overloaded.
- Cout is the object of ostream class
- Syntax: `cout<<variable`





Tokens in C ++

- The group of characters that forms an Basic Building block is called as Token.
- There are 4 types of token in C++



Identifier

- Any used defined name given to the program element by the programmer is called as identifier.
- In the C++ programming language, an identifier is a combination of alphanumeric characters, the first being a letter of the alphabet or an underline, and the remaining being any letter of the alphabet, any numeric digit, or the underline.
- Identifier are used for any variable, function, data definition etc.
- Ex: name, namee_1, 1name etc.

Rules in C++ for identifiers are:

- 1) Only Alphabets, Digits and Underscores are permitted.
- 2) Identifier name cannot start with a digit.
- 3) Key words cannot be used as a name.
- 4) Upper case and lower case letters are distinct.
- 5) Special Characters are not allowed.
- 6) Global Identifier cannot be used as Identifier.

Legal Characters:

- The set of characters in c ++ consists of alphabet, digits & special Symbols.

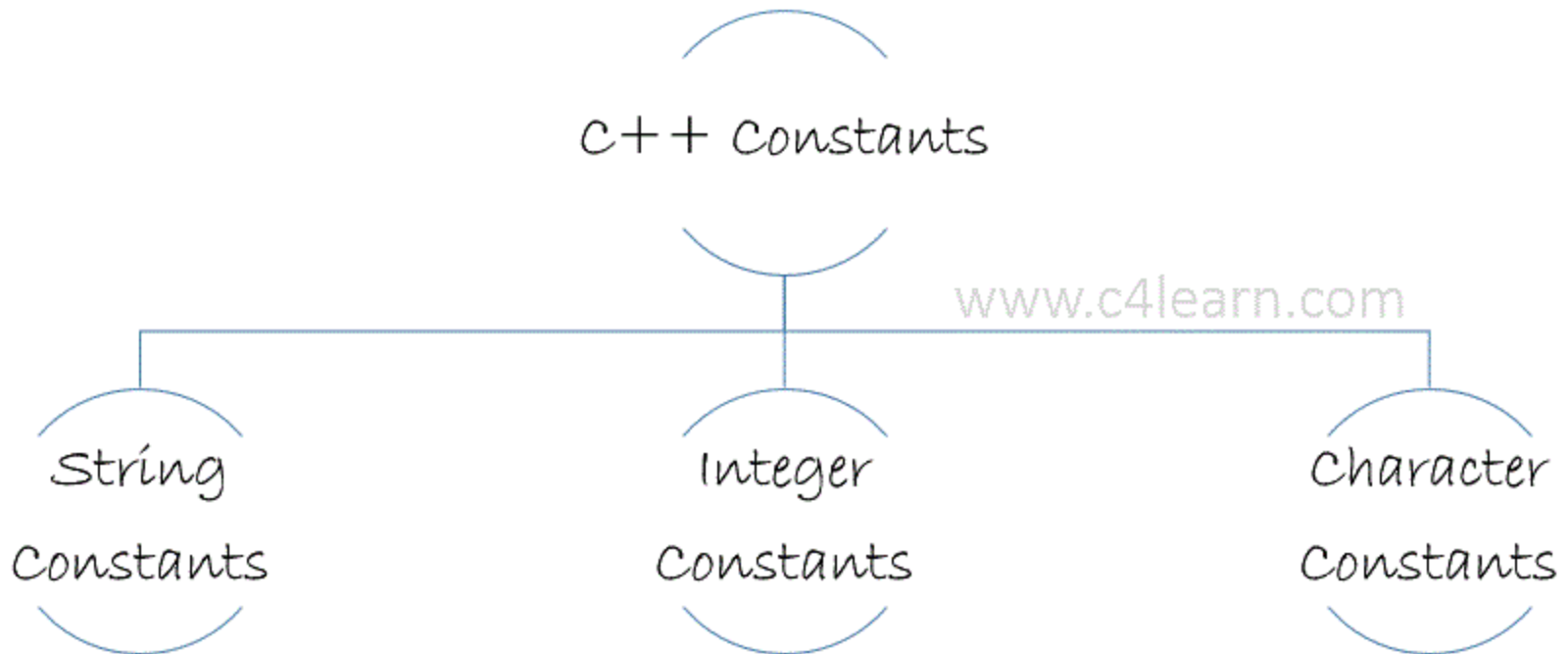
Token	Example	Definition
Alphabets	A-Z , a-z	Set of letters
Digits	0-9	Numbers
Special characters	\$, %, _, ^, :, ;	ASCII Printable characters

Keywords

- In C++, keywords are reserved identifiers which cannot be used as names for the variables in a program.
- Ex: auto, else, if, cout, cin, signed, unsigned etc.
- **Keywords cannot be used for :**
 1. Declaring Variable Name
 2. Declaring Class Name
 3. Declaring Function Name
 4. Declaring Object Name

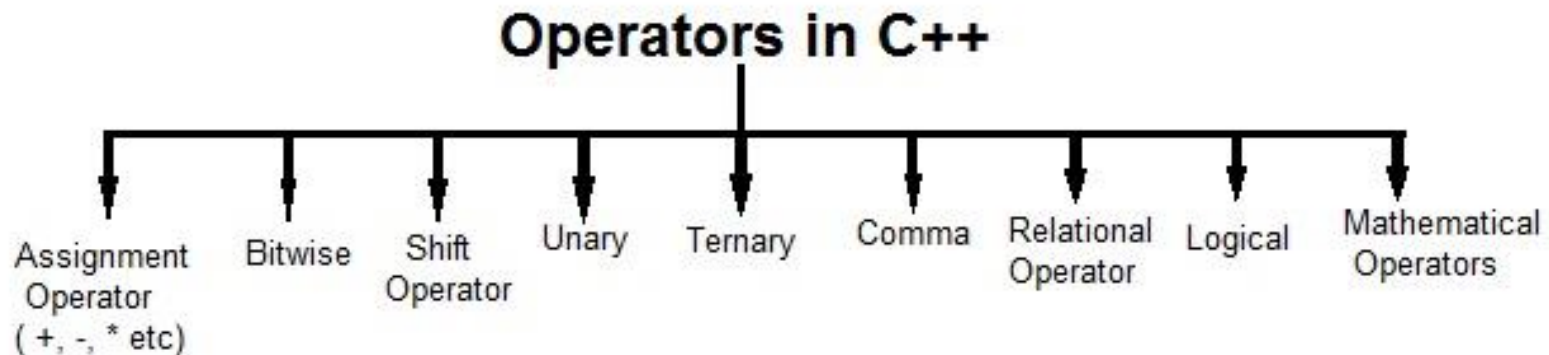
Constants:

- Constants are the items whose value cannot be changed during the program execution.



Operators:

- An operator is a symbol which tells compiler to take an action on operands and give the result.
- The Value on which operator operates is called as operands





Data types in C++

- Data types are used to define, the type of variables and contents used in the program.
- Data types can be built in or abstract.

1) Built in Data Types:

These are those data types which are predefined and are wired directly into the compiler.

- Ex: int, char etc.

Data Type	Bytes
for integral number	2
For characters	1
single precision floating point numbers	4
double precision floating point numbers	8



2) User defined or Abstract data types:

These are those data type, that user creates as a class.

In C++ these are classes ,where as in C it was implemented by structures.

Modifier



- A modifier is used to alter the meaning of the base type so that it more precisely fits the needs of various situations.
- The data type modifiers are listed here:
 - 1) signed
 - 2) unsigned
 - 3) long
 - 4) short



Some Points about Modifiers:

1) long and short modify the maximum and minimum values that a data type will hold. A plain int must have a minimum size of short.

2) **Size hierarchy :**

short int < int < long int

3) **Size hierarchy for floating point numbers is :**

float < double < long double

long float is not a legal type and there are no short floating point numbers.

Signed types includes both +ve & -ve. By default it is +ve.

Unsigned numbers are always without any sign, that is always positive.

Sizeof Operator in C ++



- sizeof is a compile-time operator, not a function.
- It is used to calculate the size (i.e. information about the amount of memory allocated) of data type ,variable & objects. It return the size in integer format.
- It can also be used to get size of user defined data types.

Syntax : sizeof(data type)

sizeof operator can be used with and without parentheses. If you apply it to a variable you can use it without parentheses.

1) `cout << sizeof(double); // will print the size of double`

2) `int x = 2;`

`int i = sizeof x;`



```
int main()
{
cout << "Size of char : " << sizeof(char) << endl;
cout << "Size of int : " << sizeof(int) << endl;
cout << "Size of short int : " << sizeof(short int) << endl;
cout << "Size of long int : " << sizeof(long int) << endl;
cout << "Size of float : " << sizeof(float) << endl;
cout << "Size of double : " << sizeof(double) << endl;
return 0;
}
```

Output: Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8

typedef Operator



typedef is a keyword used in C language to assign alternative names to existing types.

Basically it is mostly used with user defined data types, when names of data types get slightly complicated.

Syntax:

```
typedef typedef existing_name alias_name
```




Example:

```
#include<stdio.h>
int main()
{
    typedef int Number;
    Number num1 = 40,num2 = 20;
    Number answer;
    answer = num1 + num2;
    cout<< answer;
    return;
}
```

Output: 60



Decision Making statements in C++



Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met.

C++ handles decision-making by following statements:-

- 1) if statement
- 2) switch statement
- 3) conditional operator statement
- 4) goto statement

Decision making with if statement



The if statement may be implemented in different forms depending on the complexity of conditions to be tested.

The different forms are:

- 1) Simple if statement
- 2) If....else statement
- 3) Nested if....else statement
- 4) else if statement

Simple if statement:

Syntax:

```
if (condition)
{
    // body
}
```

Example:

```
int main ()
{
    int a = 10;
    if( a < 20 )
    {
        cout << "a is less than 20;" << endl;
    }
    return 0;
}
```

If.....else statement

Syntax: if(boolean expression)

```
{
```

```
// statement's will execute if the boolean expression is true
```

```
}
```

```
else
```

```
{
```

```
// statement's will execute if the boolean expression is false
```

```
}
```

```
int main ()
{
    int a = 100;
    if( a < 20 )
    {
        cout << "a is less than 20;" << endl;
    }
    else
    {
        cout << "a is not less than 20;" << endl;
    }
    cout << "value of a is : " << a << endl;
    return 0;
}
```

Nested if.....else statement:

Syntax:

```
if( expression )  
{  
    if( expression1 )  
    {  
        statement-block 1 ;  
    }  
    else  
    {  
        statement-block 2;  
    }  
    else  
    {  
        statement-block 3;  
    }  
}
```




```
int main ()
{
    int a = 100 , b = 200;
    if( a == 100 )
    {
        if( b == 200 )
        {
            cout << "Value of a is 100 and b is 200" << endl;
        }
    }
    else
    cout << "No Match"<< endl;
    return 0;
}
```

Else.....if ladder:

Syntax:

```
if(expression 1)
{
    statement-block1;
}
else if(expression 2)
{
    statement-block2;
}
else if(expression 3 )
{
    statement-block3;
}
else
    default-statement;
```



```
void main()
{
    int number;
    cout<< "Enter an integer: ";
    cin>> number;
    if ( number > 0)
    {
        cout << "You entered a positive integer: "<<number<<endl;
    }
    else if (number < 0)
    {
        cout<<"You entered a negative integer: "<<number<<endl;
    }
    else
    {
        cout<<"You entered 0."<<endl;
    }
}
```



C++ switch statement

C++ provides the multi-way decision statement which helps to decrease the complexity of the program. This is called Switch Case statement.

OR

To select particular choice among number of choices, we use Switch statement.

Switch case statement is best way to deal with the else-if nesting.

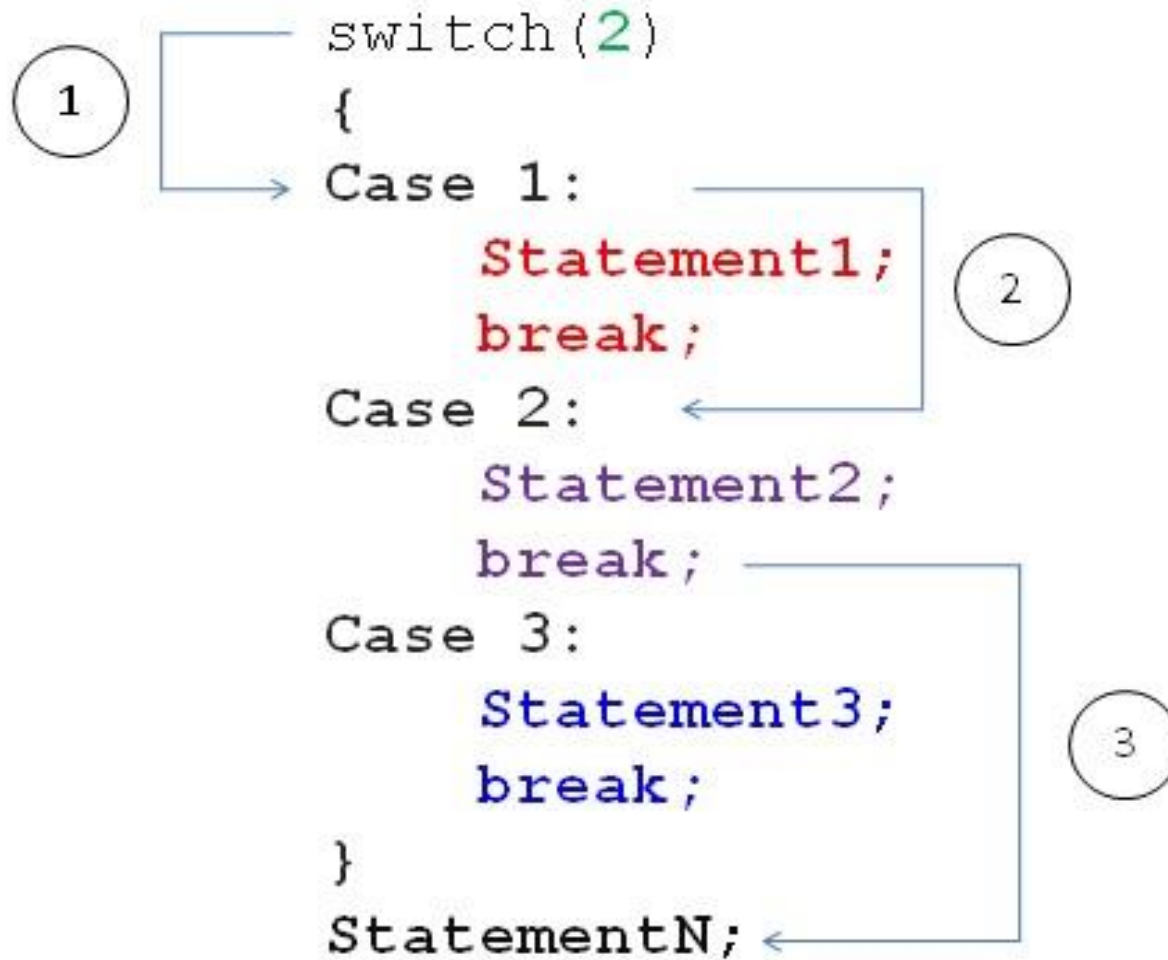
Syntax:

```
    switch(expression)
    {
        case constant-expression :
            // statement
            break;

        optional case constant-expression :
            // statement
            break;

        .
        .

        // you can have any number of case statements.
    default :
        //Optional statement's;
    }
```





```
void main ()
{
    char grade = 'C';
    switch(grade)
    {
    case 'A' :
        cout << "Excellent!" << endl;
        break;
    case 'B' :
    case 'C' :
        cout << "Well done" << endl;
        break;
    default :
        cout << "Invalid grade" << endl;
    }
    cout << "Your grade is " << grade << endl;
}
```



Loops in C ++



There may be a situation, when user need to execute a block of code several number of times. In that situation loop used.

A loop statement allows us to execute a statement or group of statements multiple times

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

1. for Loop
2. while Loop
3. Do-while Loop

1. For loop

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

Syntax:

```
for ( initialization; condition; increment )  
{  
  statement's  
}
```

Ex:



```
# 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;
}
```



While loop

A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

Syntax:

```
while(condition)
{
    statement's
}
```



```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    int a = 10;
```

```
    // while loop execution
```

```
    while( a < 20 )
```

```
{
```

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

```
    a++;
```

```
}
```

```
    return 0;
```

```
}
```



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:

```
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.



```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    int a = 10;
```

```
do
```

```
{
```

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

```
    a = a + 1;
```

```
} while( a < 20 );
```

```
    return 0;
```

```
}
```



C++ nested loops

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

Syntax:

```
for ( initialization; condition; increment )  
{  
    for ( initialization; condition; increment )  
    {  
        statement's  
    }  
    statement's  
}
```



```
int main()
{
    int i , j, rows;
    cout<<"Enter the number of rows: ";
    cin>>rows;
    for(i=1;i<=rows;++i)
    {
        for(j=1;j<=i;++j)
        {
            cout<<"* ";
        }
        cout<<"\n";
    }
    return 0;
}
```



Loop Control Statements



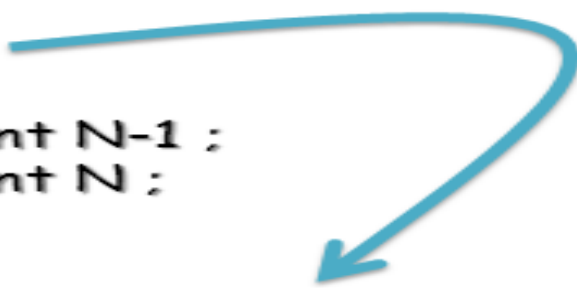
There are three types of loop control statements:

1. Break
2. Continue
3. goto

Break statement

- In C++, break statement is used for the termination of the loop statement and switch case statement.
- **Syntax:** break;

```
for ( initialization ; condition ; increment )  
{  
    Statement 1 ;  
    Statement 2 ;  
    Statement 3 ;  
    .....  
    .....  
    .....  
    break;  
  
    Statement N-1 ;  
    Statement N ;  
}  
  
OutsideStatement 1;
```

A blue arrow originates from the 'break;' statement within the for loop and points downwards and to the right, ending at 'OutsideStatement 1;', illustrating that the break statement exits the loop.

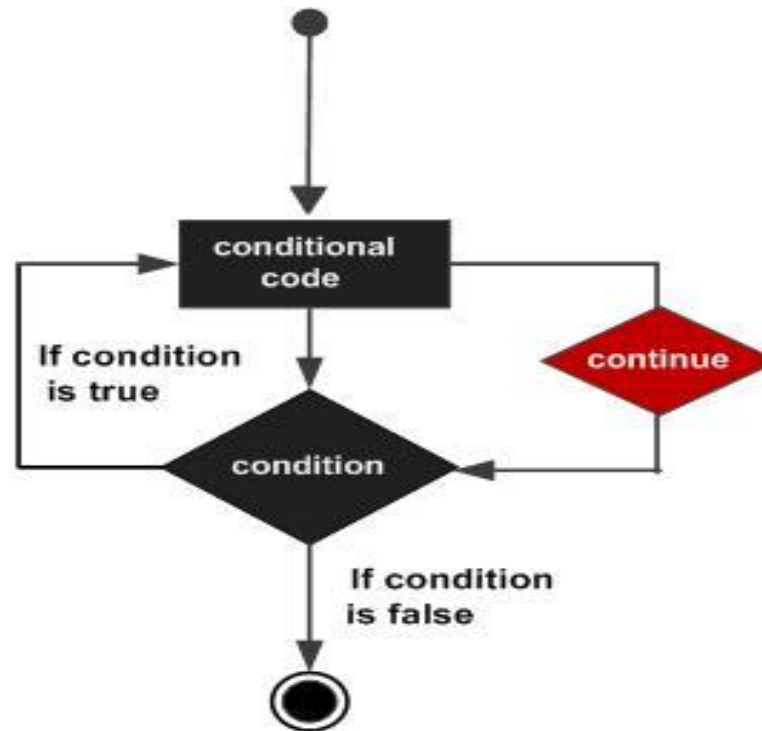


```
int main ()
{
int a = 10;
do
{
cout << "value of a: " << a << endl;
a = a + 1;
if( a > 15)
{
break;
}
}while( a < 20 );
return 0;
}
```

Continue statement



- In the break statement we can terminate the loop whenever require. Similarly we can use continue statement to skip the part of the loop.
- **Syntax:** `continue;`





```
int main ()
{
int a = 10;
do
{
cout << "value of a: " << a << endl;
a = a + 1;
if( a > 15)
{
continue;
}
}while( a < 20 );
return 0;
}
```

goto statement



A goto statement used as unconditional jump from the goto to a labeled statement in the same function.

Advantage of goto statement:

1. goto statement is used for unconditional jump.
2. goto statement makes difficult to trace the control flow of program.
3. goto statement should be avoided in order to make smoother program.

Syntax: goto label;

.

.

label: statement;



```
int main ()
{
    int a = 10;
LOOP: do
{
    if( a == 15)
    {
        a = a + 1;
        goto LOOP;
    }
    cout << "value of a: " << a << endl;
    a = a + 1;
}while( a < 20 );
return 0;
}
```



Storage classes in C ++



Storage classes are used to specify the lifetime and scope of variables.
How storage is allocated for variables and How variable is treated by compiler depends on these storage classes.

These are basically divided into 5 different types :

1. Global variables
2. Auto / Local variables
3. Register variables
4. Static variables
5. Extern variables

Global Variables:

These are defined at the starting , before all function bodies and are available throughout the program.

```
using namespace std;
```

```
int globe;// Global variable
```

```
void fun();
```

```
int main()
```

```
{
```

```
// statements;
```

```
}
```



Auto/ Local variables:

They are defined and are available within a particular scope. They are also called Automatic variable, because they come into being when scope is entered and automatically go away when the scope ends. The keyword auto is used, but by default all local variables are auto, so we don't have to explicitly add keyword auto before variable declaration. Default value of auto variable is garbage.



Register variables:

This is also a type of local variable. This keyword is used to tell the compiler to make access to this variable as fast as possible. Variables are stored in registers to increase the access speed. But you can never use or compute address of register variable and also , a register variable can be declared only within a block, that means, you cannot have global or static register variables.



Static Variables:

Static variables are the variables which are initialized & allocated storage only once at the beginning of program execution, no matter how many times they are used and called in the program. A static variable retains its value until the end of program. Static specifiers are also used in classes.

Extern Variables:

This keyword is used to access variable in a file which is declared & defined in some other file, that is the existence of a global variable in one file is declared using extern keyword in another file.



Functions in C++



A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**.

A function **declaration** tells the compiler about a function's name, return type, and parameters.

A function **definition** provides the actual body of the function.

Function are two types:

- 1) Built-in functions
- 2) User-Defined functions

Defining a Function:

```
return_type function_name ( parameter list )  
{  
  body of the function  
}
```

Return Type:

A function may return a value. The **return_type** is the data type of the value the function returns. When function data type void then it doesn't return any value.



Function Name:

This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters:

A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional. It means, a function may contain no parameters.

Function Body:

The function body contains a collection of statements that define what the function does.



Function Declarations:

A function **declaration** tells the compiler about a function name and how to call the function.

The actual body of the function can be defined separately.

A function declaration has the following parts:

```
return_type function_name( parameter list );
```

When function has parameters then Parameter names are not important in function declaration only their type is required.

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.



Calling a Function:

- To use a function, you will have to call or invoke that function.
- When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.
- To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.



```
int max(int num1, int num2); // function declaration
int main ()
{
int a = 100; int b = 200; int ret;
ret = max(a, b);           // calling a function
cout << "Max value is : " << ret << endl;
return 0;
}
int max(int num1, int num2) // function definition
{
int result;
if (num1 > num2)
result = num1;
else result = num2;
return result;
}
```



Function Arguments:

- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.
- The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
- While calling a function, there are two ways that arguments can be passed to a function:
 - 1) Call By Value
 - 2) Call By Reference



Call By Value:

- **The call by value** method , of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- By default, C++ uses call by value to pass arguments.

Ex.



```
void swap(int x, int y);
int main ()
{
    int a = 100, b = 200;
    cout << "Before swap, value of a :" << a << endl;
    cout << "Before swap, value of b :" << b << endl;
    // calling a function to swap the values.
    swap(a, b);
    cout << "After swap, value of a :" << a << endl;
    cout << "After swap, value of b :" << b << endl;
    return 0;
}
void swap(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
    return;
}
```



Call By Reference:

- The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

Ex.



```
void swap(int &x, int &y);
int main ()
{
    int a = 100; b = 200;
    cout << "Before swap, value of a :" << a << endl;
    cout << "Before swap, value of b :" << b << endl;
    swap(a, b);
    cout << "After swap, value of a :" << a << endl;
    cout << "After swap, value of b :" << b << endl;
    return 0;
}
```

```
void swap(int &x, int &y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
    return;
}
```



Default Values for Parameters:

- When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.
- This is done by using the assignment operator and assigning values for the arguments in the function definition.
- If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead.

Ex.



```
#include <iostream>
using namespace std;
int sum(int a, int b=20)
{
    int result;
    result = a + b;
    return (result);
}
int main ()
{
    int a = 100, b = 200, result;
    result = sum(a, b);
    cout << "Total value is :" << result << endl;
    return 0;
}
```



Array in C ++

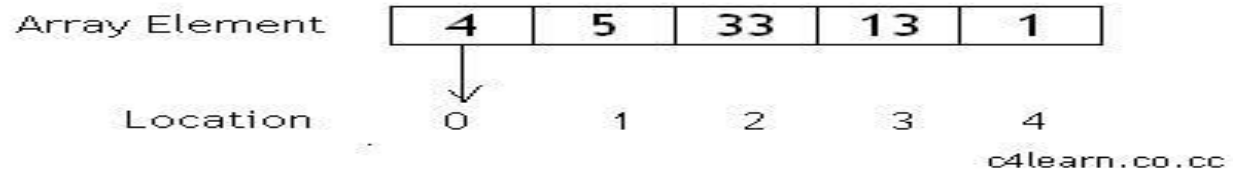
Why use Array:

- Suppose we have to store the roll numbers of the 100 students the we have to declare 100 variables named as roll1, roll2, roll3, roll100. which is very difficult job.
 - So C++ Programming gives array which have the capability to store the 100 roll numbers in the contiguous memory in 100 blocks and which can be accessed by single variable name.
- 1) An Arrays is the Collection of similar types of elements.
 - 2) All elements are stored in Contiguous memory locations & are accessed using the subscript variable.

Pictorial Look of an Array:

Here array is declared as `int a[5];`

`a[0] =4; a[1]=5; a[2] =33; a[3]=13; a[4] =1;`



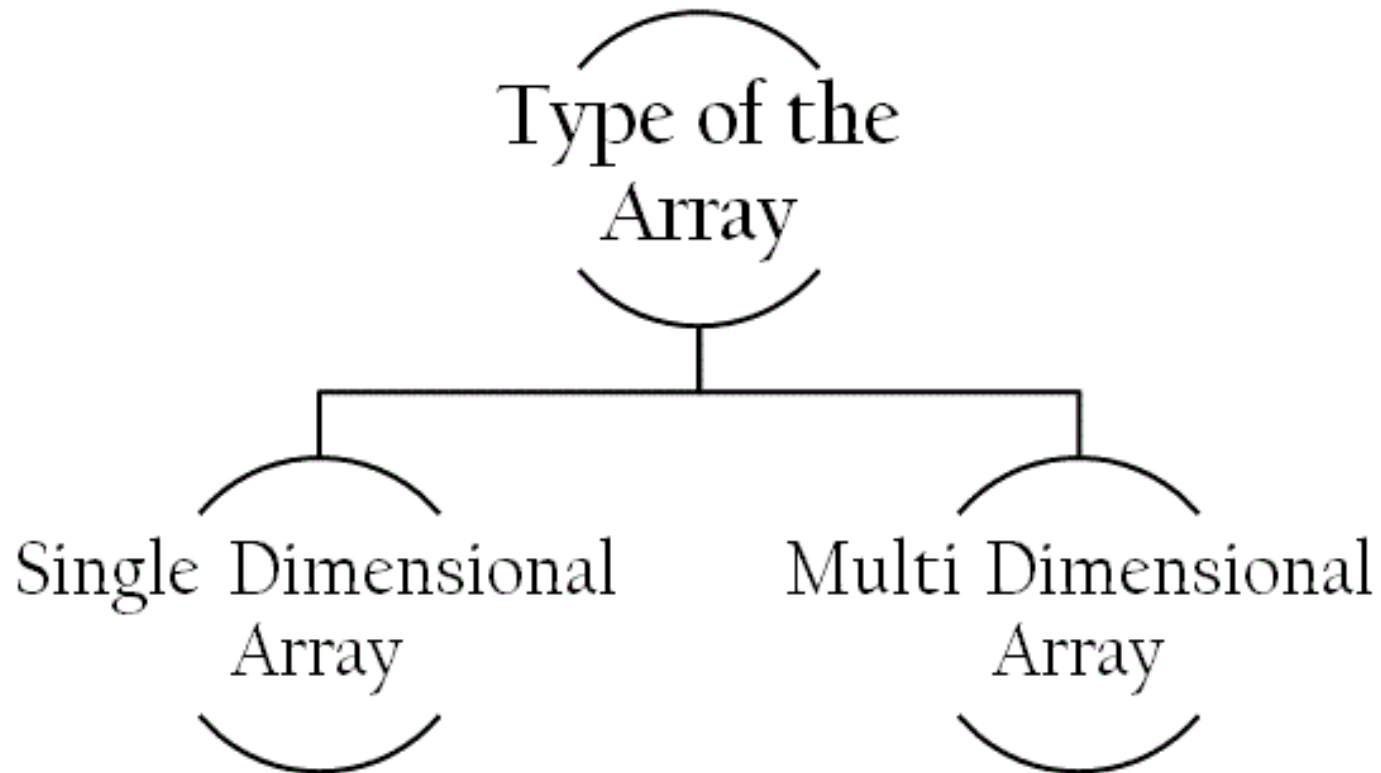
4,5,33,13,1 are actual data items & 0,1,2,3,4 are index variables.

Index or Subscript variable:

Individual data items can be accessed by the name of the array and an integer enclosed in square bracket called subscript variable / index.

Subscript Variables helps us to identify the item number to be accessed in the contiguous memory.

Types of Arrays:



1. Single Dimensional Array :

- 1) Single or One Dimensional array is used to represent and store data in a linear form.
- 2) Array having only one subscript variable is called One-Dimensional Array.
- 3) It is also called as Linear Array.

Syntax: `<data_type> <array_name> [size];`

Example:

1. `int a[3] = {2,3,4};`
2. `char c[20] = "HELLO";`

2. Multi Dimensional Array :

- 1) Array having more than one subscript variable is called Multi - Dimensional Array.
- 3) It is also called as Matrix.

Syntax: `<data_type> <array_name> [row_subscript]
[column_subscript];`

Example: 1. `int a[3][3] = {1,2,3,4,5,6,7,8,9};`

OR

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

Example:

```
int main()
{
int arr [6]= {1,2,3,4,5,6};
int i;
for(i=0;i<6;i++)
{
cout<<arr [i]<<endl;
cout<<&arr [i]<<endl;
}}
return 0;
}
```

Example :

```
int main()
{
int arr [2][3]= {1,2,3,4,5,6};
int i, j;
for(i=0;i<2;i++)
{
for(j=0;j<3;j++)
{
cout<<arr [i][j]<<endl;
cout<<&arr [i][j]<<endl;
}}
return 0;
}
```



Pointers in C++



A **pointer** is a variable whose value is the address of another variable.

Since Pointer is also a kind of variable, thus pointer itself will be stored at different memory location.

Like any variable or constant, you must declare a pointer before you can work with it.

Syntax: data_type *var-name;

Ex: int *ip; // pointer to an integer .
 double *dp; // pointer to a double .
 float *fp; // pointer to a float.
 char *ch // pointer to character

Declaration of Pointer

Syntax: data_type *var-name;

Ex: int *ip;

Explanation: Asterisk(*)

* is called as Indirection Operator.

It is also called as Value at address Operator

It indicates Variable declared is of Pointer type.

Ways of Declaration of Pointer



* can appears anywhere between Pointer_name and Data Type.

`int *p;` OR `int * p;` OR `int * p;`

Initialization of Pointer

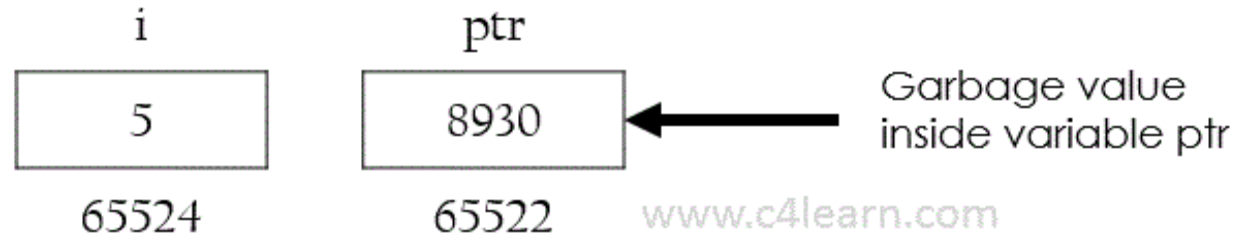


Syntax: `ptr_name = & Var_name;`

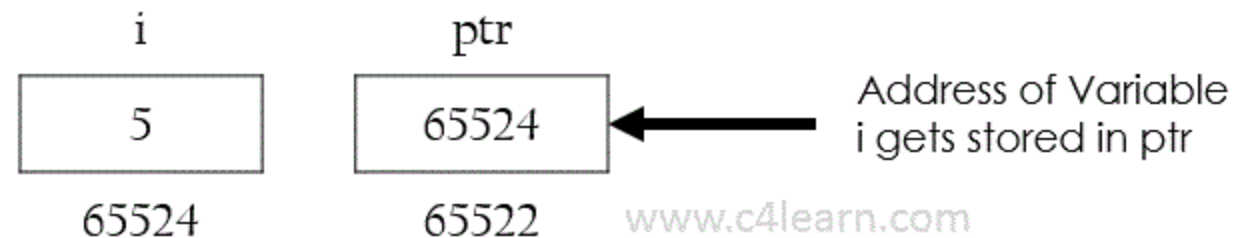
Here & is called Address Operator.

Ex:

```
void main()
{
int i, *ptr;
i=10;
```



```
ptr = &a;
cout<<ptr;
}
```





C++ pointers vs. arrays

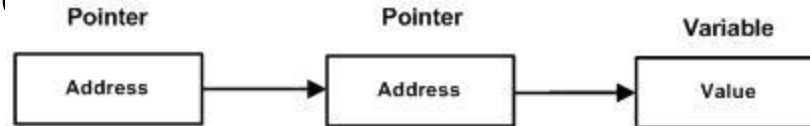
Pointers and arrays are strongly related. In fact, pointers and arrays are interchangeable in many cases. For example, a pointer that points to the beginning of an array can access that array by using either pointer arithmetic or array-style indexing.

Example:

```
const int MAX = 3;
int main ()
{
    int var[MAX] = {10, 100, 200};
    int *ptr;
    ptr = var;
    for (int i = 0; i < MAX; i++)
    {
        cout << "Address of var[" << i << "] = "; cout << ptr << endl;
        cout << "Value of var[" << i << "] = "; cout << *ptr << endl;
        ptr++;
    }
    return 0;
}
```

C++ Pointer to Pointer (Multiple Indirection)

- A pointer to a pointer is a form of multiple indirection or a chain of pointers.
- Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below



Ex: `int a;`
 `int *ptr, **ptr 1;`
 `a = &ptr;`
 `ptr 1 = &ptr;`

•

Example:

```
int main ()
{
int var, *ptr, **pptr;
var = 3000;
ptr = &var;
pptr = &ptr;
cout << "Value of var :" << var << endl;
cout << "Value available at *ptr :" << *ptr << endl;
cout << "Value available at **pptr :" << **pptr <<
"
```



String in C++

A string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello".

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

OR

```
char greeting[] = "Hello";
```



Example:

```
int main ()  
{  
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\\0'};  
cout << greeting << endl;  
return 0;  
}
```

String Functions:

S.N.	Function & Purpose
1	strcpy(s1, s2); Copies string s2 into string s1.
2	strcat(s1, s2); Concatenates string s2 onto the end of string s1.
3	strlen(s1); Returns the length of string s1.
4	strcmp(s1, s2); Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1.
6	strstr(s1, s2); Returns a pointer to the first occurrence of string s2 in string s1.



Example:

```
#include <iostream>
#include <cstring>
using namespace std;
int main ()
{
    char str1[10] = "Hello";
    char str2[10] = "World";
    char str3[10];
    int len ;
    strcpy( str3, str1); // copy str1 into str3
    cout << "strcpy( str3, str1) : " << str3 << endl;
    strcat( str1, str2); // concatenates str1 and str2
    cout << "strcat( str1, str2): " << str1 << endl;
    len = strlen(str1); // total length of str1 after concatenation
    cout << "strlen(str1) : " << len << endl;
    return 0;
}
```