

Chapter 3

C++ Languages Constructs

6hours



Preprocessor Directive:

The line belonging with hash (#) sign are called preprocessor directive. They are not program statements as they are not terminated by semi-colon. They communicate with preprocessor. #include , #define are two preprocessor, we have used so far.

e.g.

```
#include <iostream>
```

The #define directive cause a symbolic name to be defined as a macro and associates it with a sequence of tokens, called the body of macro.

General format of simple macro:

```
#define macro_name sequence_of_tokens
```

```
#define N 10
```



User-Defined Constant `const`:

Both C and C++ provides a mechanism to define symbolic constants using the keyword '`const`'. This indicates that the value does not change during program execution. Since we cannot change the value of the constant we must initialize it.

e.g.

```
const int max=1000;
```

```
const float a=4.5;
```

```
const int arr[] ={1,2,3,4};
```

After declaring constant ,they cannot be changed within scope.



In C++, Constant are also used to specify array bounds as:

```
const int max_size=5;
```

```
int marks[max_size];
```

In case of C++, every 'const' should be initialized during declaration. Whereas, in case of ANSI C, initialization is not mandatory . For un-initialized identifier , it automatically initializes to 0.

In ANSI C, 'const' values are global. They are visible outside the file in which they are declared . In C++, a 'const' value is local to the scope where it is declared.



Type Conversion And Promotion Rules:

We can convert one basic datatype into another . There are two types of conversion.

1. Implicit Type Conversion(Automatic Conversion): It does not requires any operator and it is performed by compiler itself. In an expression, when we try to evaluate two or more data items of different type , then one type should be converted into another(i.e. lower type to higher type)before final result is calculated . The implicit conversion is also known as promotion or widening or arithmetic conversion.



char

Lower Type

short

int

long

float

double

long double

Higher Type




e.g.

```
int main(){  
    int m=15;  
    float x=3.1,y;  
    y= m*x;
```

// where, x has type float has high ranking and m has type int is converted into float.

```
    cout << " \n Value of Y: "<<y<< endl;  
}
```

Output:46.5



2. Type Cast or Explicit Conversion: The conversion of one datatype into another , performed by programmer as per need is explicit conversion.

Syntax:

`(type) expression //C style.`


or,

`type (expression) //older C++ style.`

e.g. `float x;`


`(int) x;`

If value of `x=3.99`, then after cast , its value will be 3



```
int main(){  
    int x=5;  
    float y;  
    y= (float) x/2;  
    cout << "Value of Y: " << y <<endl ;  
    return 0;  
}
```

Output: 2.5.



Although these casts still works, ANSI C++ adds the following new cast operators:

1. **static_cast:** This is used for any standard conversion of datatype . It can also be used to cast a base class pointer into a derived class pointer.

Its general form is:

`static_cast<type>(object)`

e.g. `int m=10;`

`double x=static_cast<double>(m);`

`char ch = static_cast<char>(m);`



e.g. `#include<iostream.h>`

```
int main(){
```

```
float q;
```

```
int num1=7,num2=9;
```

```
q=num1/num2;
```

```
cout<<"Without casting : q="<<q; //displays 0 which is wrong result.
```


```
q=static_cast<float>(num1)/num2;
```

```
cout <<"After casting : q="<<q; //right result
```

```
return 0;
```

```
}
```

The problem with old style is that the cast operator is not distinctly seen in the code . Also , it is difficult to locate the cast location by making use of search facilities available in code editors because the same keywords are used in both type conversion and data declaration. The cast operator like `static_cast` is distinct and easy to spot because it is only used in type conversion.

- 
2. `const_cast` : `const_cast<type>(object)`
 3. `dynamic_cast` : `dynamic_cast<type>(object)`
 4. `reinterpret_cast` : `reinterpret_cast<type>(object)`



Manipulators:

Manipulators are special type of function used to modify the output as user's requirements . Some of the commonly used manipulators are endl , setw() ,setfill() ,and setprecision() . The header file <iomanip.h> must be included to use the standard manipulators.

1. **The endl manipulator**: This Manipulator is used to insert new line character (i.e. change line in output) . Its function is same as '\n' in C program.

```
#include <iostream.h>
```

```
#include<iomanip.h>
```

```
void main(){
```

```
cout << "Welcome"<<endl<<"Khwopa";
```

```
}
```



Output:

Welcome

Khwopa


2. The setw() Manipulator: This manipulator sets the minimum field width on output. It prints the number or string within the field specified in the argument.

```
#include<iostream.h>
```

```
#include<iomanip.h>
```

```
int main(){
```


```
    long pop1=5425678 , pop2=47000;
```

```
cout<< setw(8) <<
"LOCATION"<<setw(12)<<"POPULATION"<<endl
<<setw(8)<<"Patan"<<setw(12)<<pop1<<endl
<<setw(8)<<"Butwal"<<setw(12)<<pop2<<endl;
return 0;
}
```

Output:

LOCATION	POPULATION
Patan	5425678
Butwal	47000




3. The setfill() Manipulator: This is used for filling the blank fields . The character specified in the setfill() argument of the manipulator is used to fill the blank fields.

```
#include<iostream.h>
#include<iomanip.h>
void main(){
int num1=12345, num2=2356;
cout<<endl<<"num1="<<setw(20)<<setfill('*')<<num1<<endl;
cout<<"num2="<<setfill('$')<<setw(8)<<num2<<endl;
return 0;
}
```

Output:

```
num1=*****12345
```

```
num2=$$$$2356
```



4. The setprecision() Manipulator: This is used with floating point numbers. It is used to specify the number of digits to be displayed after decimal point.

```
#include<iostream.h>
#include<iomanip.h>
void main(){
float num=70.6776;
cout<<"Num with precision three="<<setprecision(3)<<num<<endl;
}
```

Output:

Num with precision three=70.678



Enumeration:

C++ provides a mechanism to create user defined data type called enumerated data type which provides a way of referring names for constants.

The dictionary meaning of 'enumeration' is a numbered list. We can create our own data type ; where every possible value is defined as a constant (i.e. called an enumerator).

The enumerated types are declared via the 'enum' keyword. The enum keyword automatically assigns list of data items with integer value 0,1,2,3 and so on.



Syntax:

```
enum enum_datatype_name
```

```
{
```

```
    Enumerator1;
```

```
    Enumerator2;
```

```
    .....
```

```
    .....
```

```
    Enumerator n;
```

```
};
```

e.g.

```
#include<iostream.h>
```


```
int main(){
```



```
enum Color{  
    RED,  
    GREEN,  
    BLUE,  
    WHITE  
};  
Color c;  
c=BLUE;  
cout<<"Your color code is:"<<c;  
return 0;  
}
```

Output:

Your color code is:2



Defining an enumerated type does not allocate any memory. When a variable of the enumerated type is declared (such as c in the above e.g.), memory is allocated for that variable at that time . The enum variables are the same size as an int variable.

(To Do List For Students):

1. Character Set and Tokens (Literals , Identifiers ,Keywords , Operators and Punctuators)
2. Variable declaration and expression.
3. Statements
4. Datatype
5. Condition and looping .
6. Array , Pointer and String ,Union.



Dynamic Memory Allocation

With new and delete:

The process of allocating and freeing memory at run-time is known as dynamic memory allocation . This reserves the memory required by a program and free the memory once the reserved space is utilized.

C++ provides 'new' operator for dynamic memory allocation and 'delete' operator for dynamic memory de-allocation.

1. new Operator:

- To request the dynamic memory , new operator is used . The new operator obtains memory at run-time from the memory heap from the operating system and returns the address of the obtained memory.

- The memory reserved by new in any scope remains as it is even after the program control goes out of the scope.

- Syntax:

```
pointer_variable =new Datatype;
```

e.g. 1.

```
int *p;  
p=new int;
```


It is used to allocate memory to contain one single element of particular type.

e.g.2.

```
int *p;  
p=new int[10];
```

It is used to assign a block (an array) of elements of type int. The system dynamically assigns space for ten elements of type int and return a pointer to the first elements of a sequence, which is assigned to pointer p. Therefore, p points to a valid block of memory with space for ten elements of type int.

The first element pointed by p can be accessed either with the expression p[0] or *p . The second element can be accessed either with p[1] or *(p+1) and so on.



2. **delete Operator**: When we allocate memory using 'new' at run-time ,the memory is reserved even after it is not needed or after the scope of the variable ends. When the reserved memory is no longer needed, it should be freed so that the memory becomes available again for other requests of dynamic memory. We use 'delete' operator to free allocated memory.

Q. What happen when reserved memory is not freed after use?

Ans: We allocate memory according to our need . After using the memory if we leave it as unuse then it is called the **memory leak**. As we allocate more and more memory , the program might then run out of memory space for data manipulation and eventually result is abnormal program termination. So, when the dynamically allocated memory is no longer needed it has to be destroyed to prevent memory leakage.



The delete operator is used as follows:

`delete pointer_variable; //release a single dynamic variable.`

`Delete [] pointer_variable; //release dynamically created array.`

e.g.

```
int *p;
```

```
p=new int;
```

```
.....
```

```
delete p;
```

E.g. for multiple element (array)

```
int *p;
```

```
p=new int[5];
```

```
.....
```

```
delete [ ] p;
```

The equivalent of delete statement is

```
free(pointer_variable)
```



Q. WAP to find the sum and average of the number by using new and delete operator.

Ans: #include <iostream.h>

```
int main(){
```

```
int n , i , *p , tot=0;
```

```
float avg;
```

```
cout<<"How many numbers:";
```

```
cin>>n;
```


```
p=new int[n]; //allocate memory for array.
```

```
cout<<"Enter elements";
```

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

```
cin>>p[i];
```

(continue.....)



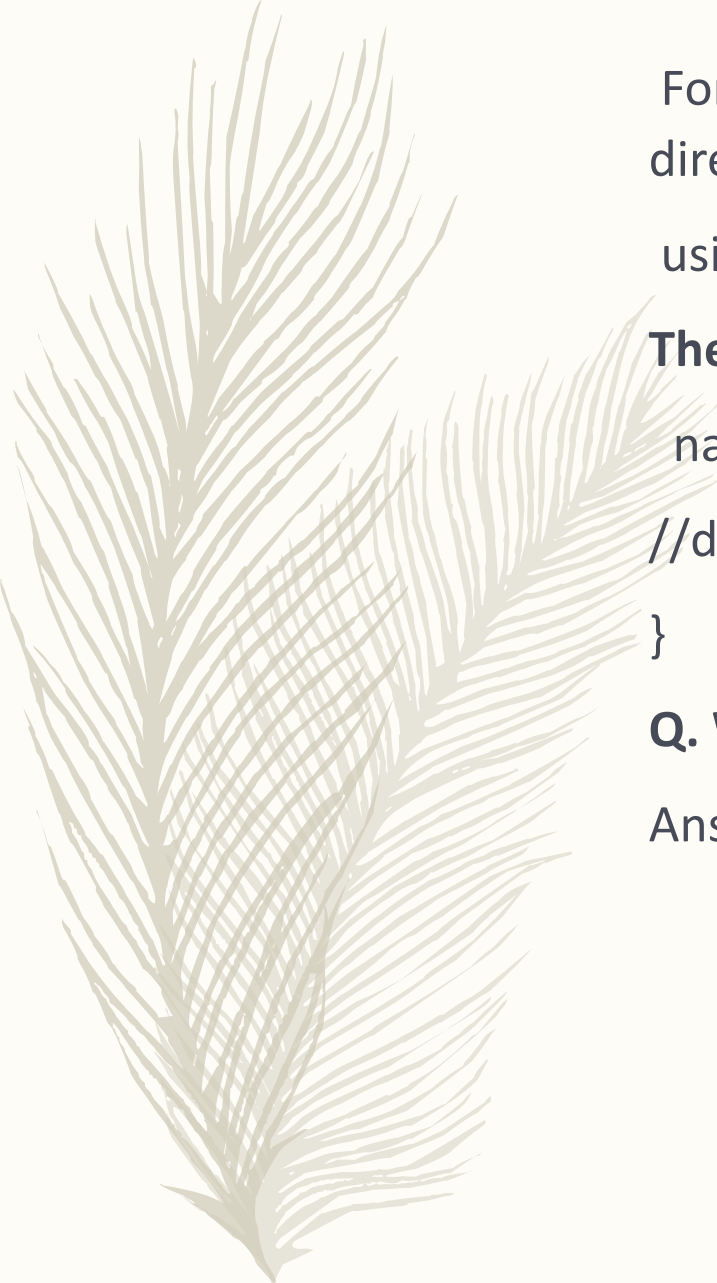
```
tot+ = p[i];    // tot=tot+ p[i];  
    }  
    avg= static_cast <float>(tot)/n;  
    cout<<"Total="<<tot<<endl;  
    cout<<"Average="<<avg;  
    delete [ ] p;  
    return 0;  
}
```



Namespace:

The namespace mechanism is used for logical grouping of variables , classes and functions in C++. In general, we can't define two classes or functions with same name in a program. But sometimes there may be situation where functions, variables for classes in one program will match with existing names of variables , functions and classes such that there will be conflict in identifier's name.

C++ uses namespace to avoid the conflicts. The namespace provides scope and we can define functions , variables and classes within the scope . Thus, a namespace is a container for variables , functions ,classes and other identifiers and allows the disambiguation of homonym(i.e. having same name) identifiers residing in different scopes.



For using identifiers defined in the namespace scope, we must include the using directive, like

```
using namespace std;
```

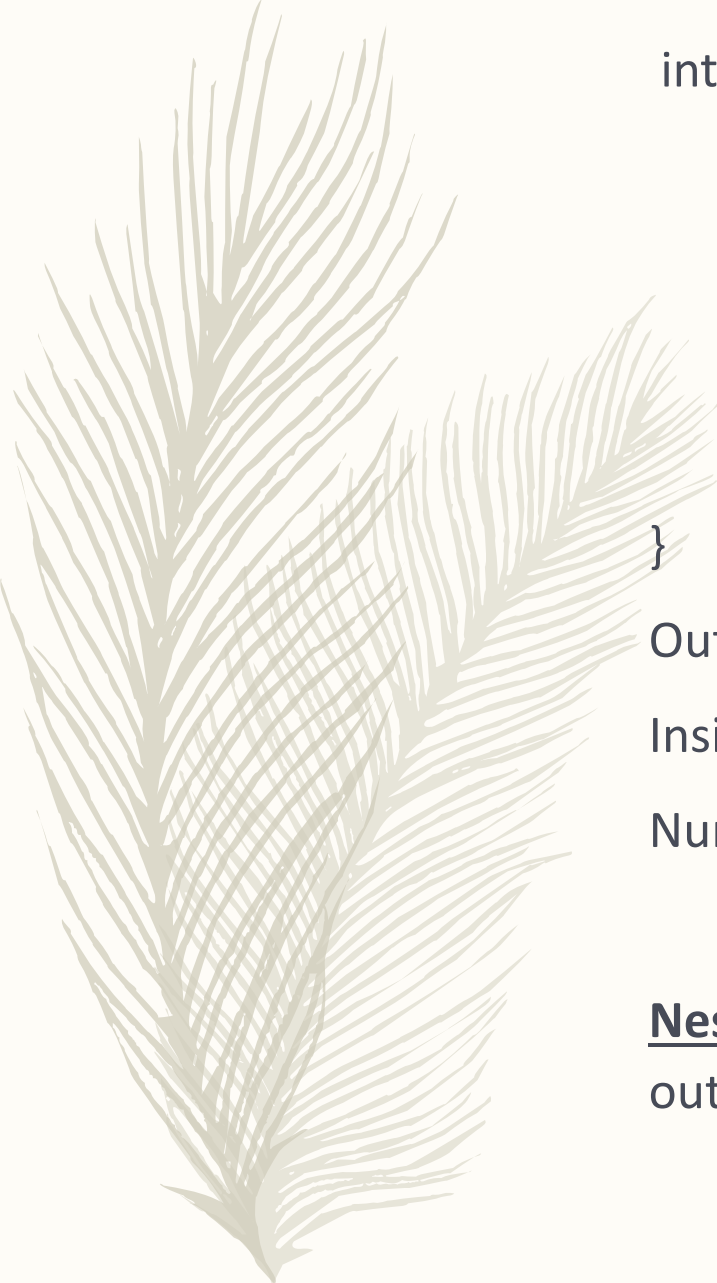
The syntax for defining the namespace is :

```
namespace namespace_name {  
    //declaration of variables, classes, functions etc.  
} //no semicolon at the end.
```

Q. WAP to illustrate the use of namespace.

Ans: #include<iostream.h>

```
namespace number{  
    int num = 10;  
}
```




```
int main(){  
    int num=20;  
    cout<<"Inside main:num="<<num;  
    cout<<"Num within namespace="<<number::num;  
    return 0;  
}
```

Output:

Inside main:num=20

Num within namespace=10

Nested Namespace: It is the namespace within another namespace. They are outer namespace and inner namespace.

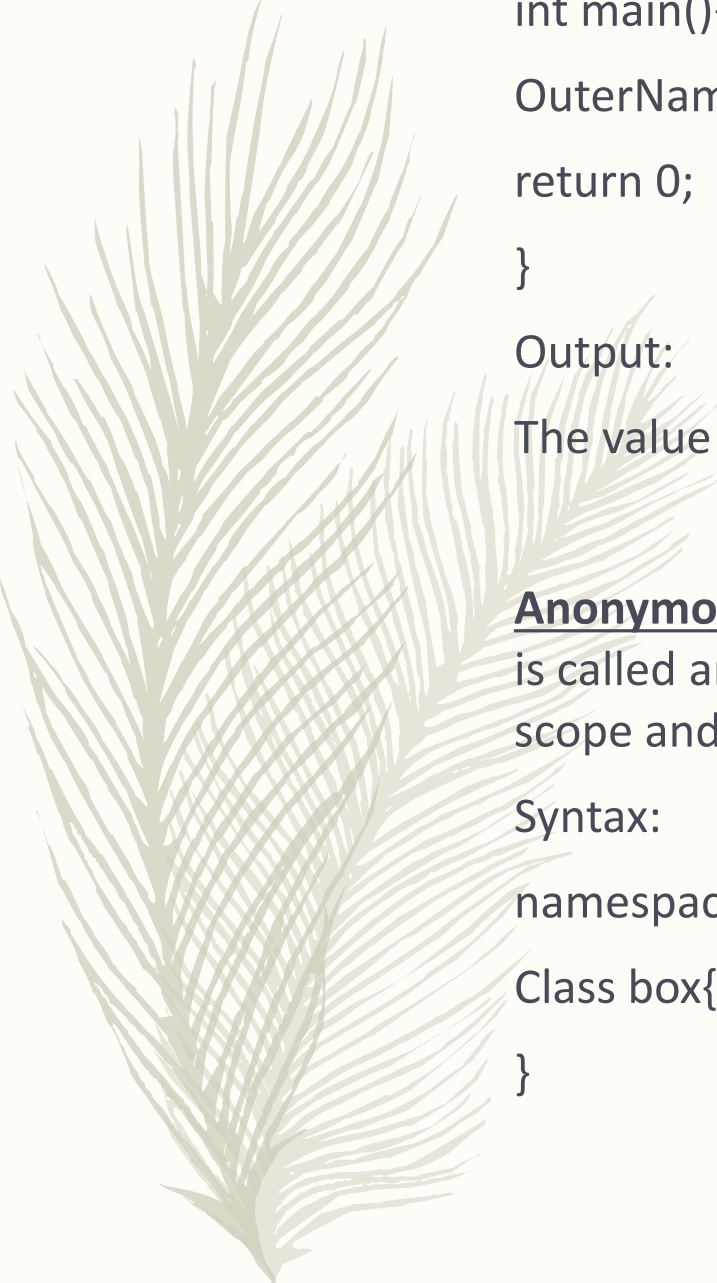


```
namespace Outer_namespace
{
namespace Inner_namespace
{
}
}
```

Q. WAP to illustrate the use of nested namespace.

Ans: #include<iostream.h>

```
namespace OuterNamespace{
    int num=10;
    namespace innerNamespace{
        void display(){
            std::cout<<"The value of num is:"<<num;
        }
    }
}
```



```
int main(){  
OuterNamespace::InnerNamespace::Display();  
return 0;  
}
```


Output:

The value of num is :10

Anonymous Namespace: We can define namespace without name . Such type of namespace is called anonymous or unnamed namespace. Unnamed namespace members occupy global scope and are accessible in all scope.

Syntax:

```
namespace {  
Class box{  
}
```

For each unnamed namespace , the compiler generates a unique name, which differs from every other name in the program.

```
#include<iostream.h>
```

```
namespace{
```

```
    int i=4;
```

```
    int m;
```

```
}
```

```
int main(){
```


```
    cout<<"i="<<i;
```

```
    m=100;
```

```
    cout<<"m="<<m;
```

```
    return 0;
```

```
}
```



Here, the variables i and m have been defined within unnamed namespace. As unnamed namespace occupy global scope, they can be accessed within main() function also.

Accessing identifiers defined within Namespace:

1. Use of using keyword.
2. Use of scope resolution operator.

Use of 'using' keyword to access all members through using directive:

```
#include <iostream>

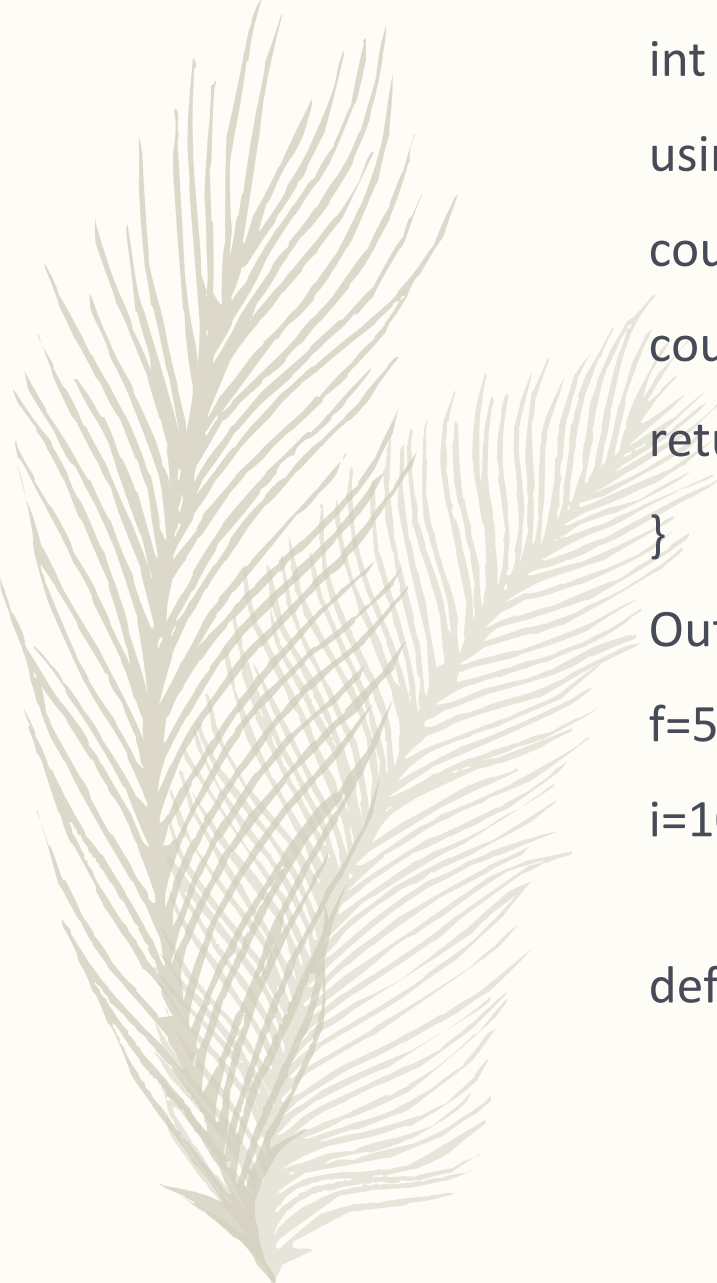
using namespace std;

namespace First{

float f=5.5;

int i=100;

}
```



```
int main(){  
    using namespace First;  
    cout<<"f=\t"<<f<<"\n";  
    cout<<"i=\t"<<i<<"\n";  
    return 0;  
}
```


Output:

f=5.5

i=100

In this e.g. , the keyword 'using' is being used such that all members defined within the specified namespace(i.e. First) can be accessed normally.

Use of scope resolution operator to access members:



```
#include <iostream>

using namespace std;

namespace First{

    float f=4.56;

    int i=100;

}

int main(){

    cout<<"f="<<First::f<<"\n";

    cout<<"i="<<First::i<<"\n";

    return 0;

}
```

Output: f=4.56
i=100



Type Of Function Call:

The argument in function can be passed in two ways :pass by value and pass by reference(address).


1.Function call by value(Pass by value): When values of actual arguments are passed to a function as arguments, it is known as function call by value. In this call, the value of each actual argument is copied into corresponding formal argument of the function definition. The content of the argument in the calling function are not altered , even if they are changed in the called function.

e.g. `#include <iostream.h>`

```
void swap(int , int);
```

```
void main(){
```

```
int a =99, b=89;
```



```
cout<<"Before function calling , a and b are:"<<a<<b;
swap(a , b);
cout<<"After function calling, a and b are:"<<a<<b;
}
void swap(int x , int y){
int temp;
temp=x;
x=y;
y=temp;
cout<<"The values within functions are:"<<x<<y;
}
```

Before function calling , a and b are: 99 89

The values within functions are:89 99

After function calling, a and b are: 99 89

Reference Variable:

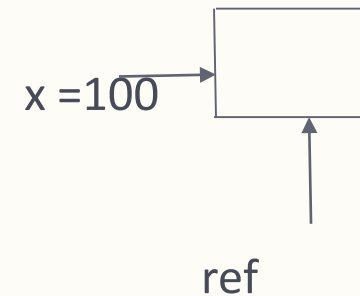
It is just alias (i.e. other name) for variable. It is declared using &(ampersand) operator . Defining reference variable is a mechanism to define a second name for a variable that we can use to read or modify the original data stored in that variable.

```
#include <iostream.h>


void main(){
int x=100;
int &ref=x;
cout<<"x="<<x<<"\t Ref="<<ref;}
```

Output:

x=100 Ref=100



Here x and ref refers same memory.



2. **Function call by Reference(Pass by reference)**: In this type of function call, the address of variable or argument is passed to the function as argument instead of actual value of variable. So, the variable passed as arguments during the function call are changed by the called function.


```
#include <iostream.h>
```

```
void swap(int &,int &);
```

```
int main(){
```

```
int a=5, b=6;
```

```
cout<<"Before swapping : a="<<a<<"and b="<<b<<endl;
```



```
swap(a , b);  
cout<<"After swapping: a="<< a<<"and b="<<b<<endl;  
return 0;  
}  
void swap(int &x, int &y){  
int temp;  
temp=x;  
x=y;  
y=temp;  
}
```

In this program, formal arguments x and y becomes alias of actual arguments a and b during function call. When variable x and y are interchanged , actual argument a and b are also interchanged.




Function Overloading:

- Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.
- In C++, two functions can have the same name if the number and/or type of arguments passed is different.
- When a function name is overloaded with different jobs it is called Function Overloading.
- In Function Overloading “Function” name should be the same and the arguments should be different.
- Function overloading can be considered as an example of polymorphism feature in C++.

[Note: In C++, functions can not be overloaded if they differ only in the return type.]

[Question For You: Functions that cannot be overloaded in C++ are??]




```
#include <iostream>

using namespace std;

// function with 2 parameters.....
void show (int x, float y) {
    cout << "Integer number: " << x << " and float number: " << y << endl;
}

// function with (float type) single parameter.....
void show(float x) {
    cout << " Float number: " << x << endl;
}

// function with (int type) single parameter.....
void show(int x) {
    cout << "Integer number: " << x << endl;
}
```



```
int main() {  
    int a = 5;  
    float b = 5.5f;  
    // call function with int type parameter  
    show(a);  
    // call function with float type parameter  
    show(b);  
    // call function with 2 parameters  
    show(a, b);  
    return 0;  
}
```




Default Argument:

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn't provide a value for the argument with a default value.


Some important points to know while performing default argument :

- Default arguments are overwritten when calling function provides values for them.
- Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.

- 
- During calling of function, arguments from calling function to called function are copied from left to right.
 - Once default value is used for an argument in function definition, all subsequent arguments to it must have default value. It can also be stated as default arguments are assigned from right to left.

– **Sample Program:**

```
#include<iostream>
using namespace std;
int add (int a, int b, int c =0, int d =0)
{
    return (a + b + c + d);
}
```

```
int main()
{
    cout << add(20, 30) << endl;
    cout << add(20, 30, 40) << endl;
    cout << add(20, 30, 40, 50) << endl;
    return 0;
}
```



inline Function:

- An inline function is a function that is expanded in-line when it is invoked thus saving time.
- The compiler replaces the function call with the corresponding function code that reduces the overhead of function calls.
- To eliminate the time of calls to small functions, C++ proposes a new function called inline function.


Syntax:

```
inline function-header  
{  
function-body  
}
```



Things to remember,

- inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining.
- Compiler may not perform inlining in such circumstances like:
 - 1) If a function contains a loop. (for, while, do-while)
 - 2) If a function contains static variables.
 - 3) If a function is recursive.
 - 4) If a function return type is other than void, and the return statement doesn't exist in function body.
 - 5) If a function contains switch or goto statement.



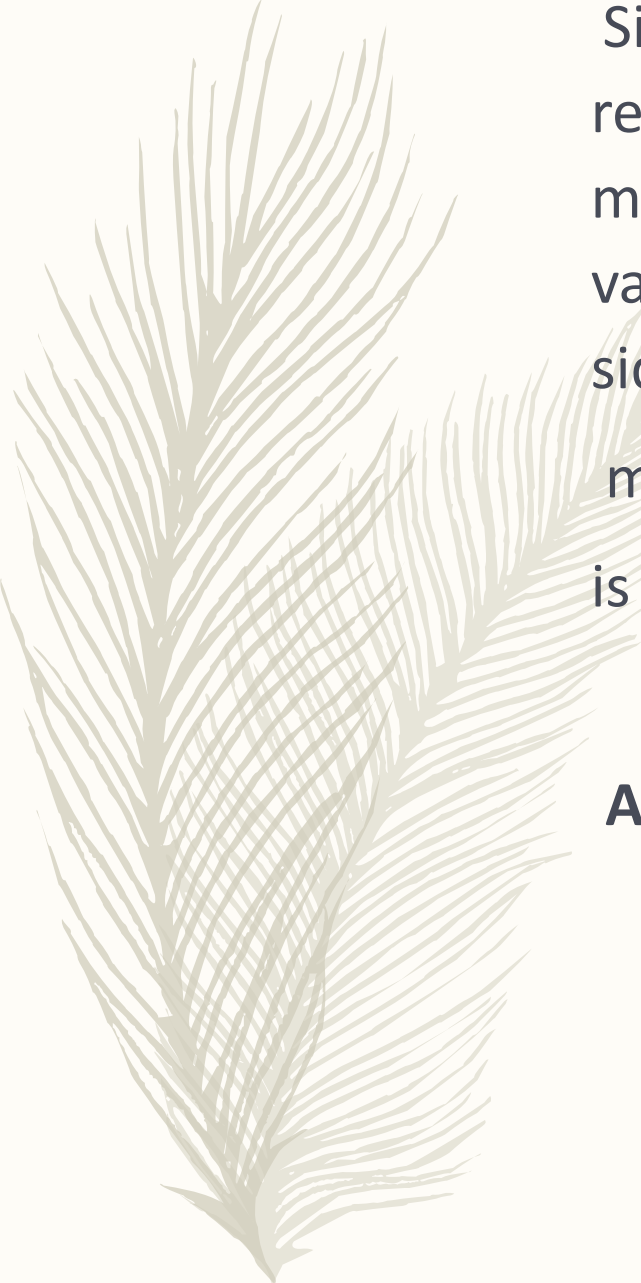
```
/*Simple Inline Function Program*/  
inline void add(){  
    int x ,y , z;  
    cout<<"Enter the value of x and y :"<<endl;  
    cin>>x>>y;  
    z= x + y;  
    cout<<"Value of z is: "<<z<<endl;  
}  
int main() {  
    add();  
    return 0;  
}
```



Return By Reference:

A function can also return a reference. Consider the following function.

```
int & max(int &x, int &y){  
    if(x>y)  
        return x;    //return reference  
    else  
        return y;  
}
```



Since, the return type of `max()` is `int &`, the function returns reference to `x` and `y` (and not the value). Then a function call such as `max(a , b)` will yield a reference to either `a` or `b` depending on their values. This means that this function call can appear on the left-hand side of an assignment statement. That is, the statement

```
max(a , b)= -1;
```

is legal and assigns `-1` to `a` if it is larger, otherwise `-1` to `b`.

Assignment: const Reference Arguments(e.g. `const int &x`).



const Reference Arguments:

- One of the major disadvantages of pass by value is that all arguments passed by value are copied into the function parameters. When the arguments are large structs or classes, this can take a lot of time.
- References provide a way to avoid this penalty. When an argument is passed by reference, a reference is created to the actual argument (which takes minimal time) and no copying of values takes place. This allows us to pass large structs and classes with a minimum performance penalty.
- However, this also opens us up to potential trouble. References allow the function to change the value of the argument, which is undesirable when we want an argument be read-only. If we know that a function should not change the value of an argument, but don't want to pass by value, the best solution is to pass by const reference.
- As we know that a const reference is a reference that does not allow the variable being referenced to be changed through the reference. Consequently, if we use a const reference as a parameter, we guarantee to the caller that the function will not change the argument



Structure in C++

[Already Done.

Include this topic in your assignment.]