C++ Function

Introduction

```
void show(); /* function declaration*/
main()
{
.....
.....
show(); // function calling
.....
.....
}
void show() // function definition
{
.....
..... // function body
.....
}
```

Main Function:

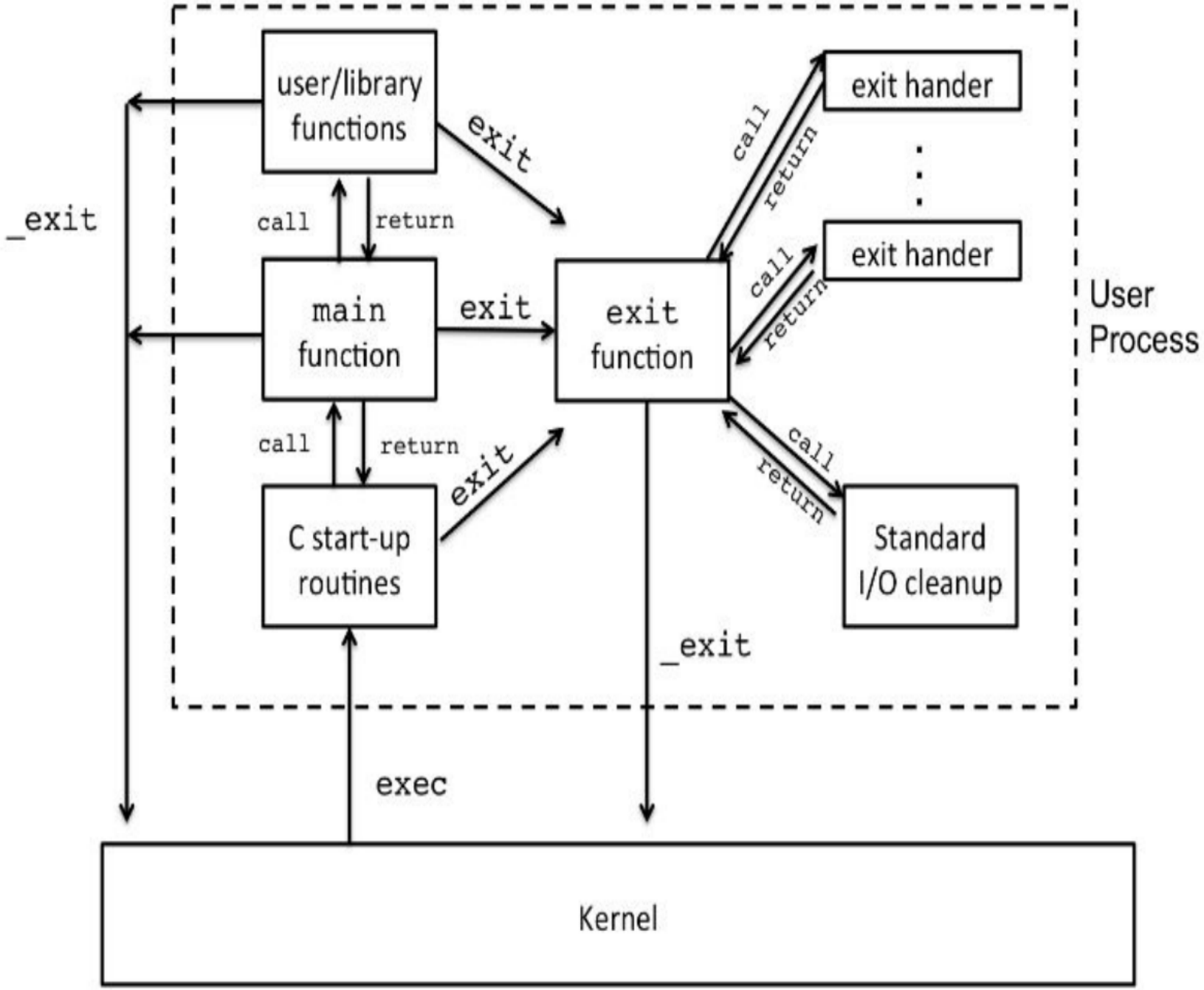
C language doesn't specify any return type for the main() function which is the starting point for the execution of a program.

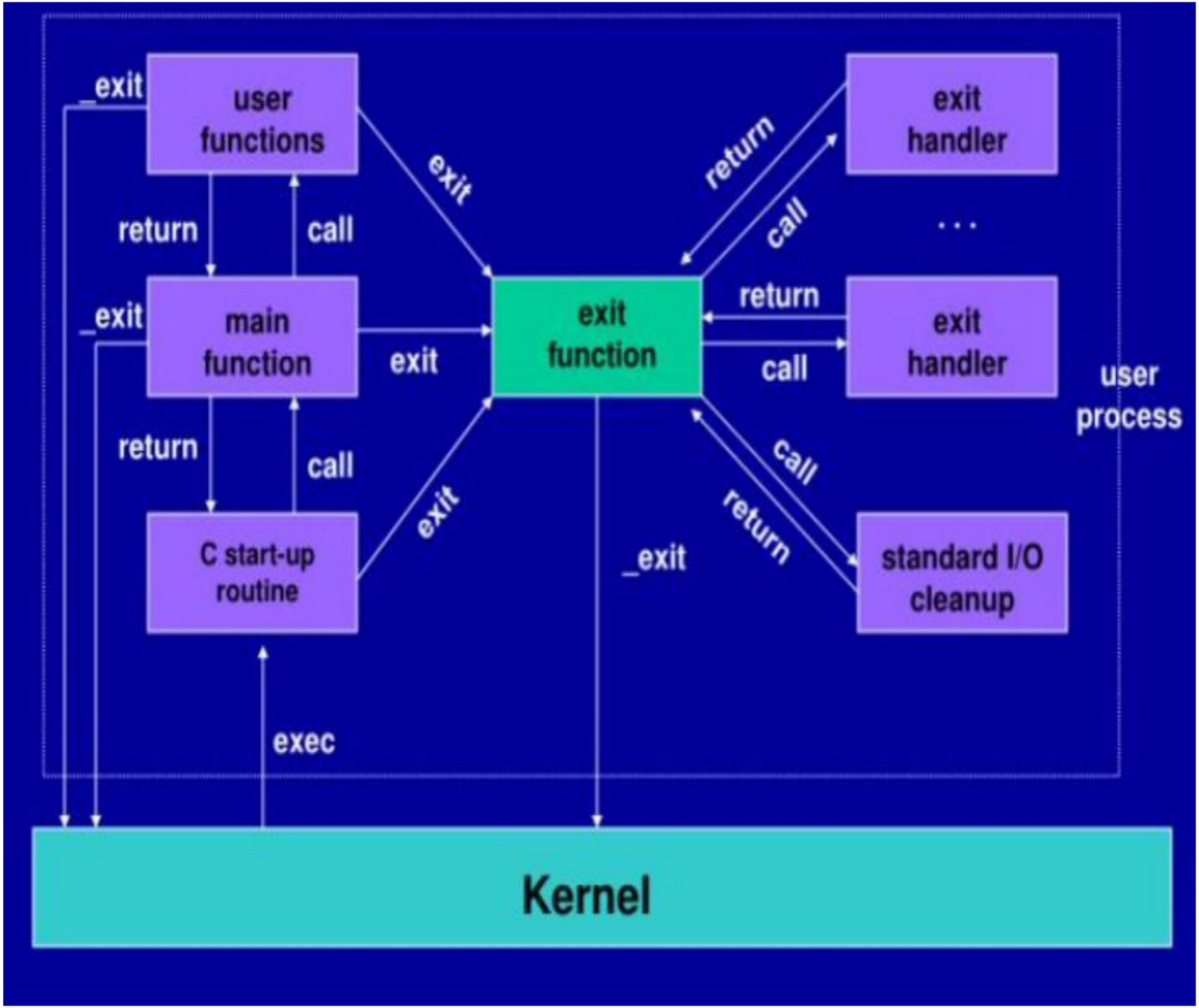
```
main()
{
// main program statements
}
```

In C++ language the main() returns a value of type int to the operating system.

```
int main();
int main(int argc, char * argv[]);
```

```
int main()
{
.....
.....
return 0;
}
```


user/library functions
exit
exit hander
call
return
_exit
call
return
exit hander
main function
exit
exit function
call
return
User Process
call
return
exit
call
return
C start-up routines
Standard I/O cleanup
_exit
exec
Kernel


_exit
user functions
exit
return
call
exit handler
. . .
_exit
main function
exit
exit function
return
call
exit handler
user process
return
call
exit
C start-up routine
_exit
call
return
standard I/O cleanup
exec
Kernel

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

int main(int argc, char *argv[])
{
   cout<<"\n Number of arguments: "<< argc;
   for (int i = 0; i < argc; ++i)
   {
      cout<<"\n Argument "<<i<<": "<< argv[i];
   }
   return 0;
}
```

C++ FUNCTION PROTOTYPING

- With function prototyping, a template is always used when declaration and defining a function.
- while c++ makes the prototyping essential, ANSI C makes it optional, perhaps, to preserve the compatibility with classic C

type function_name(arguments list);

Examples:

```
float volume(int x, float y, float z);
float volume(int x, float y,z); // it is illegal
float volume(int,float,float); // names of the arguments optional.
void display();
```

- In the definition of function, names are required because arguments must be referenced inside the function

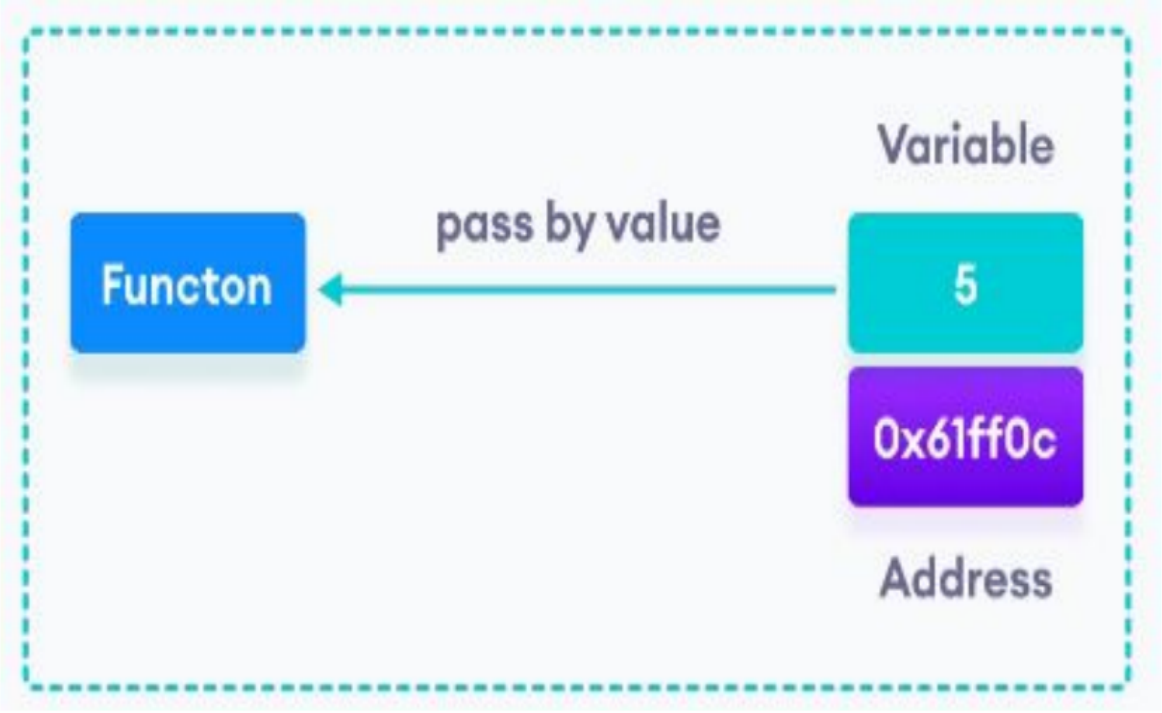
```
float volume(int a, float b, float c)
{
float v=a*b*c;
.....
.....
}
```

Write a C++ function to calculate simple interest

C++ Call by value

- When we pass a value from the calling function to the called function, it is known as call by value.

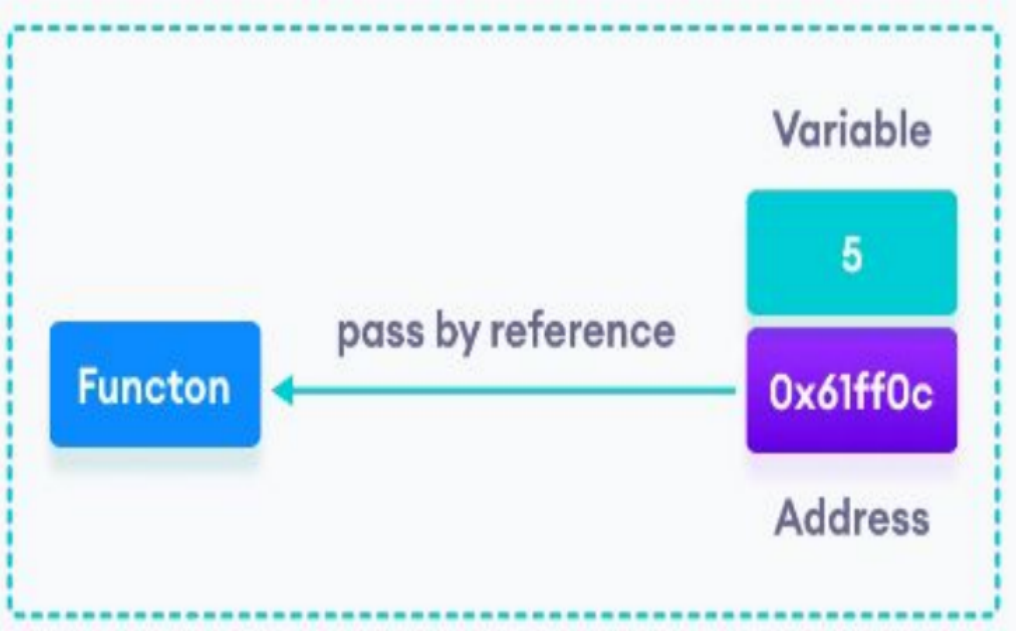
- Any changes make to the parameters inside the function do not effect the actual arguments.



C++ call by reference

- The call by reference is a method of passing arguments to a function

- Copies the address of argument into the formal parameter



Passing by reference without pointers

```
#include <iostream>
using namespace std;
void swap(int&,int&);
int main()
{
	int a=10,b=20;
	cout<<"\nVALUES BEFORE SWAPPING :\n";
	cout<<"A:"<< a<<",B:"<< b;

	swap(a,b);
	cout<<"\nVALUES AFTER SWAPPING :\n";
	cout<<"A:"<< a<<",B:"<< b;
	cout<<"\n";
	return 0;
}
```

```
// function definition
void swap(int &x,int &y)
{
	int t;
	t=x;
	x=y;
	y=t;
}
```

C++ call by reference

Passing by reference using pointers

```
#include <iostream>
using namespace std;
void swapn( int *num1, int *num2 )
{
int temp ;
temp = *num1 ;
*num1 = *num2 ;
*num2 = temp ;
}
```

```
int main( )
{
int n1 = 50, n2 = 30 ;
cout<<"Before swapping:";
cout<<"\n n1 is"<<n1;
cout<<"\n n2 is"<<n2;

/*calling swap function*/
swapn( &n1, &n2 );

cout <<"\n After swapping:";
cout <<"\n n1 is" <<n1;
cout <<"\n n2 is" <<n2;
return 0;
}
```

Return by reference

- A function can also return a reference.

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

max(a,b) = -1; is legal

- Returns a reference to x or y(not the values)

```
#include <iostream>
using namespace std;
int & max(int &x, int &y)
{
    if (x>y)
        return x;
    else
        return y;
}
int main()
{
int a=10, b=15;
max(a,b) = -1;
cout<<" value of a= "<< a<< " and b = "<<b;
return 0;
}
```

```
using namespace std;
int& largestNumber(int&,int&,int&);
int main(){
    int a,b,c;
    a=10,b=22,c=21;

    cout<<"Largest number is :"<< largestNumber(a,b,c)<< endl;
    return 0;
}
int & largestNumber(int &x,int &y,int &z){
    if(x > y && x > z)
        return x;
    else if(y > x && y> z)
        return y;
    else
        return z;
}
```

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

int vals[] = {10, 12, 33, 24, 25};

int& setValues( int i ) {
  return vals[i];
}

int main ()
{
   cout << "Value before change" << endl;
  for ( int i = 0; i < 5; i++ )
  {
    cout << "vals[" << i << "] = ";
    cout << vals[i] << endl;
  }
  setValues(1) = 20;
  setValues(3) = 70;

  cout << "Value after change" << endl;
  for ( int i = 0; i < 5; i++ )
  {
    cout << "vals[" << i << "] = ";
    cout << vals[i] << endl;
  }
  return 0;
}
```

When returning a reference, be careful that the object being referred to does not go out of scope.

So it is not legal to return a reference to local var.

But you can always return a reference on a static variable.

static makes variable last until program ends, not function end.

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

int& func() {
 int x = 5;
//static int x = 5;

   return x;
}

int main () {
   int n;
  n = func();
  cout<<"the value is:  "<<n;
  return 0;
}
```

```
#include <iostream>
class Foo {
  public:
        int& bar()
        {
              return obj;
         }
  private:
         int obj = 42;
};

int main()
{
   Foo foo;
   foo.bar() = 43;
   std::cout << foo.bar() << '\n';
}
```

foo -- and therefore foo::obj -- exists for longer than the function itself;

it exists for the entire scope of main.

CONST ARGUMENT

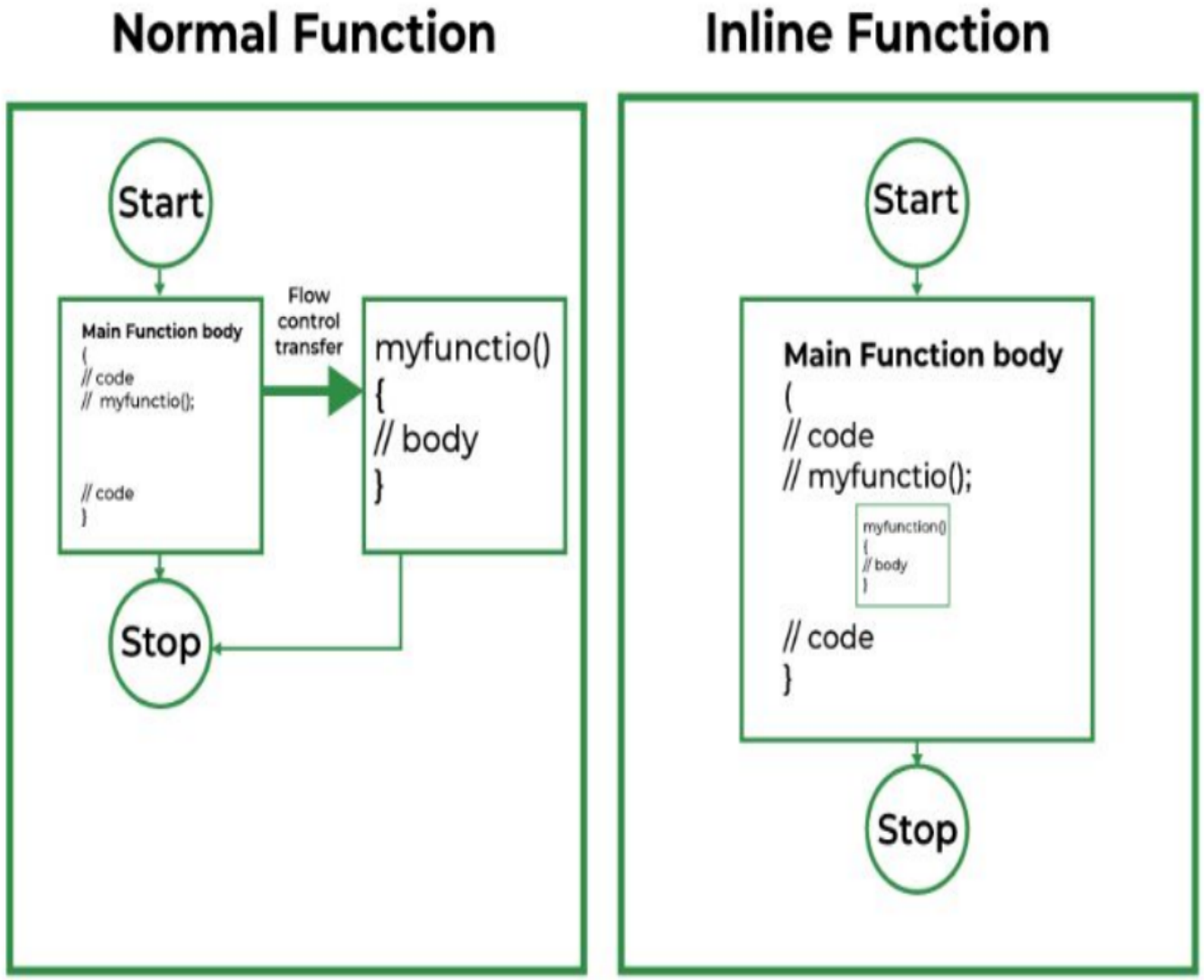
```
int strlen(const char *p);
int length(const string &s);
```

- The qualifier **const** tells the compiler that the function should not modify the argument
- Compiler generates error if it is violated
- This type of declaration is significant only when we pass arguments by reference or pointers.

Inline Function

- An inline function is a function that is expand in line when it is invoked
- To Reduce the cost of calls to small functions
- Similar to macros expansion in C; but macros are not really functions and therefore, The usual error checking does not occur during compilation
- Some of the conditions where inline expansion may not work

1. for functions returning values, a switch, if a loop, or a goto exists
2. for functions not return any values, if a return statement exists
3. if functions contain static variable
4. if inline functions are recursive


Normal Function
Start
Main Function body
{
// code
// myfunctio();
// code
}
Flow control transfer
myfunctio()
{
// body
}
Stop
Inline Function
Start
Main Function body
{
// code
// myfunctio();
myfunction()
{
// body
}
// code
}
Stop

```
#include <iostream>
using namespace std;
inline float mul(float x,float y)
{
   return (x*y);
}

inline double div(double p, double q)
{
   return (p/q);
}

int main()
{
float a=34.67;
float b=9.67;
cout<<mul(a,b)<<"\n";
cout<<div(a,b)<<"\n";
return 0;
}
```

Making an Outside Function Inline

```
class product
{
    ……
    ……
    public:
        void getdata(int x, float y); // function declaration
};
inline void product :: getdata(int x, float y) // function definition outside the class
{
    count = x;
    price = y;
}
```

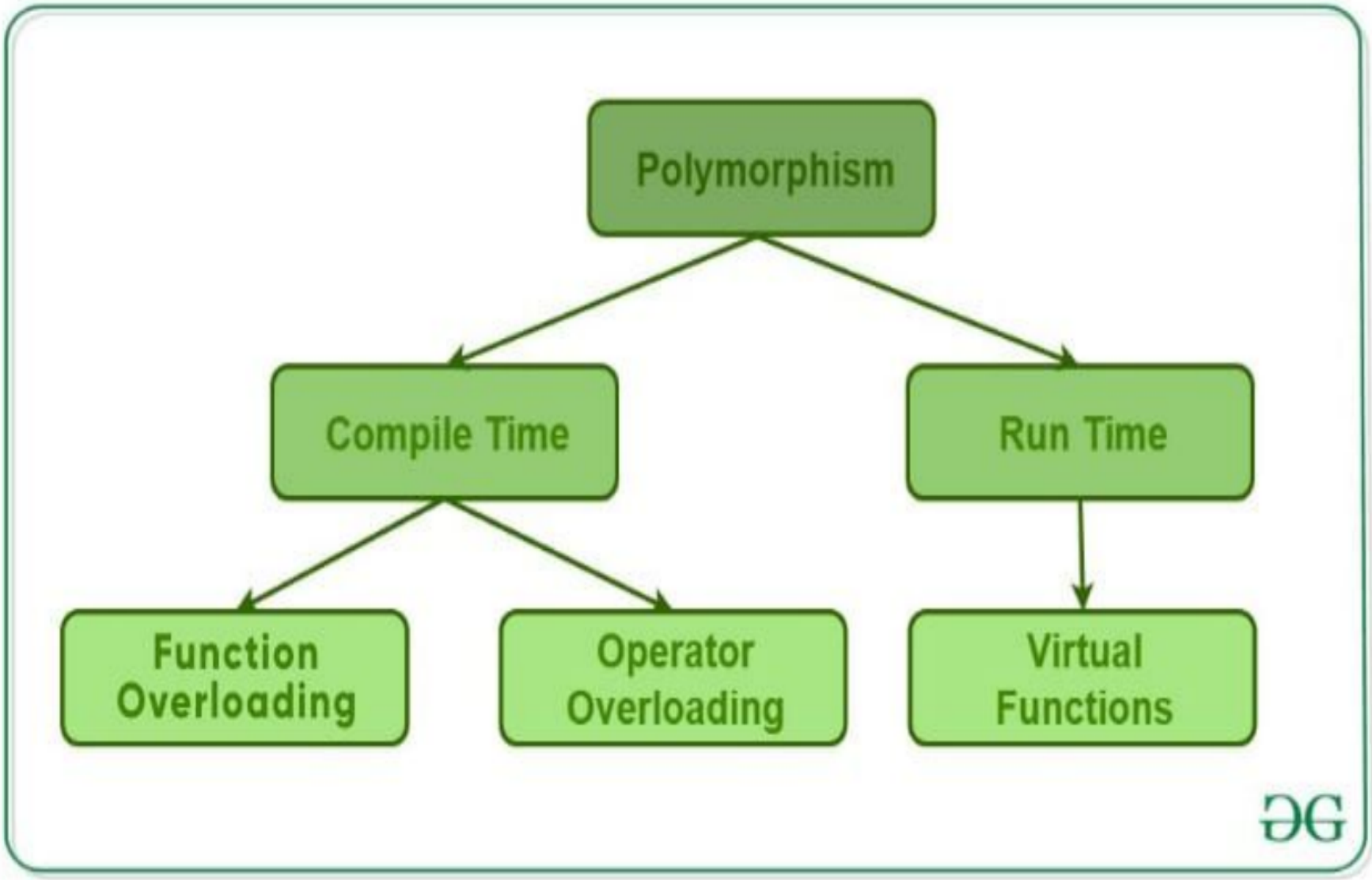
Polymorphism

In school
behave like a student

In bus
behave like a
passenger

In home
behave like a son

In shopping mall
behave like a customer


Polymorphism
Compile Time
Run Time
Function Overloading
Operator Overloading
Virtual Functions

Function overloading

- We can use the same functions names to create functions that performs a variety of different tasks.

- This is known as ***function polymorphism*** in Object oriented programming.

- One function name but with different argument (parameters) list.

- The correct function to be invoked is determined by checking the number of arguments and type of the arguments but not on the function type.

```
// declarations
int add(int a, int b); //prototype 1
int add(int a, int b, int c); // prototype 2
double add(double x, double y) // prototype 3
double add(int p,double q); // prototype 4
double add(double p, int q); // prototype 5

// FUNCTION calls
std::cout<<add(7,10); // for uses prototype 1
std::cout<<add(67,34,23); // for uses prototype 2
std::cout<<add(34.5,23.6); // for uses prototype 3
std::cout<<add(12,34.7); // for uses prototype 4
std::cout<<add(34.7,12); // for uses prototype 5
```

Let's Try…. !!

Write a C++ program that demonstrates function overloading by defining three overloaded functions named volume() to compute:

- The volume of a cube (given the side length as an integer).
- The volume of a cylinder (given the radius as a double and height as an integer).
- The volume of a rectangular box (given the length as a long, width as an integer, and height as an integer).

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

// declarations (prototypes)
int volume(int);
double volume(double, int);
long volume(long,int,int);

int main()
{
      std::cout<<volume(10)<<"\n";
      std::cout<<volume(2.5,8)<<"\n";
      std::cout<<volume(100L,75,15)<<"\n";
      return 0;
}
```

```
// Function definitions
int volume(int s) //cube
{
return (s*s*s);
}

double volume(double r, int h) // cylinder
{
return (3.14519*r*r*h);
}

long volume(long l, int b, int h) // rectangular box
{
return (l*b*h);
}
```

Default Arguments

- C++ allows us to call a function without specifying all its arguments
- The function assigns a ***default value*** to the parameter which does not have a matching argument in the function call
- ***default values*** are specified when the function is declared

Default Arguments

```
float amount(float principal, int period, float rate=0.15);
value=amount(5000,7); //so here one arguments missing
value=amount(5000,5,0.12); // no missing argument
```

- Only the trailing arguments can have default at the time of calling from right to left.

- We can't provide a default value to a particular argument in the middle of an argument list

```
int mul(int i, int j=5, int k=10); // it is legal
int mul(int i=5,int j); // this is illegal
int mul(int i=0, int j, int k=10); // this is illegal
int mul(int i=2,int j=6; int k=90); // it is legal
```

- Advantage of providing the default arguments are

1. we can use default arguments to add new parameters to the existing functions.

2. Default arguments can be used to combine similar function into one

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

int sum(int x, int y, int z = 0, int w = 0)
{
   return (x + y + z + w);
}

int main()
{
   cout << sum(10, 15) << endl;

   cout << sum(10, 15, 25) << endl;

   cout << sum(10, 15, 25, 30) << endl;
   return 0;
}
```

C++ Friend Functions

- Private members can't be accessed from outside the class
- C++ allows the common function to be made friendly with both the classes, thereby allowing function to have access to the private(hidden) data of these classes

```
class ABC
{
…….
…….
…….
public:
……..
……..
friend void xyz(); // declaration
          // is a friend function
};
```

- The function declaration should be preceded by the keyword friend.
- The function is defined anywhere in the program like a normal C++ function.
- The function definition does not use either the keyword friend or the scope operator ::
- A function can be declared as a friend in any number of classes.

- A friend function possesses certain special characteristics

1. It is not in the scope of the class to which it has been declared as friend.

2. Since it is not in the scope of the class, it can't be called using the object of that class.

3. It can be invoked like a normal function without the help of any object.

4. Unlike member functions, it can't access the member names directly and has to use an object name and dot membership operator with each member name.

5. It can be declared either in the public or the private part of a class without affecting its meaning.

6. Usually, it has the objects as arguments.

```cpp
#include <iostream>
using namespace std;
class Box
{
   private:
      int length=0;
   public:
       friend int printLength(Box); //friend function

};
int printLength(Box b)
{
   b.length += 10;
   return b.length;
}
int main()
{
   Box b;
   cout<<"Length of box: "<< printLength(b)<<
endl;
   return 0;
}
```

Let’s solve…

Define friend function. Explain how one can bridge two classes using friend Function. Write a C++ program to find the sum of two numbers using friend Function. Assume two variables are present in two different class.

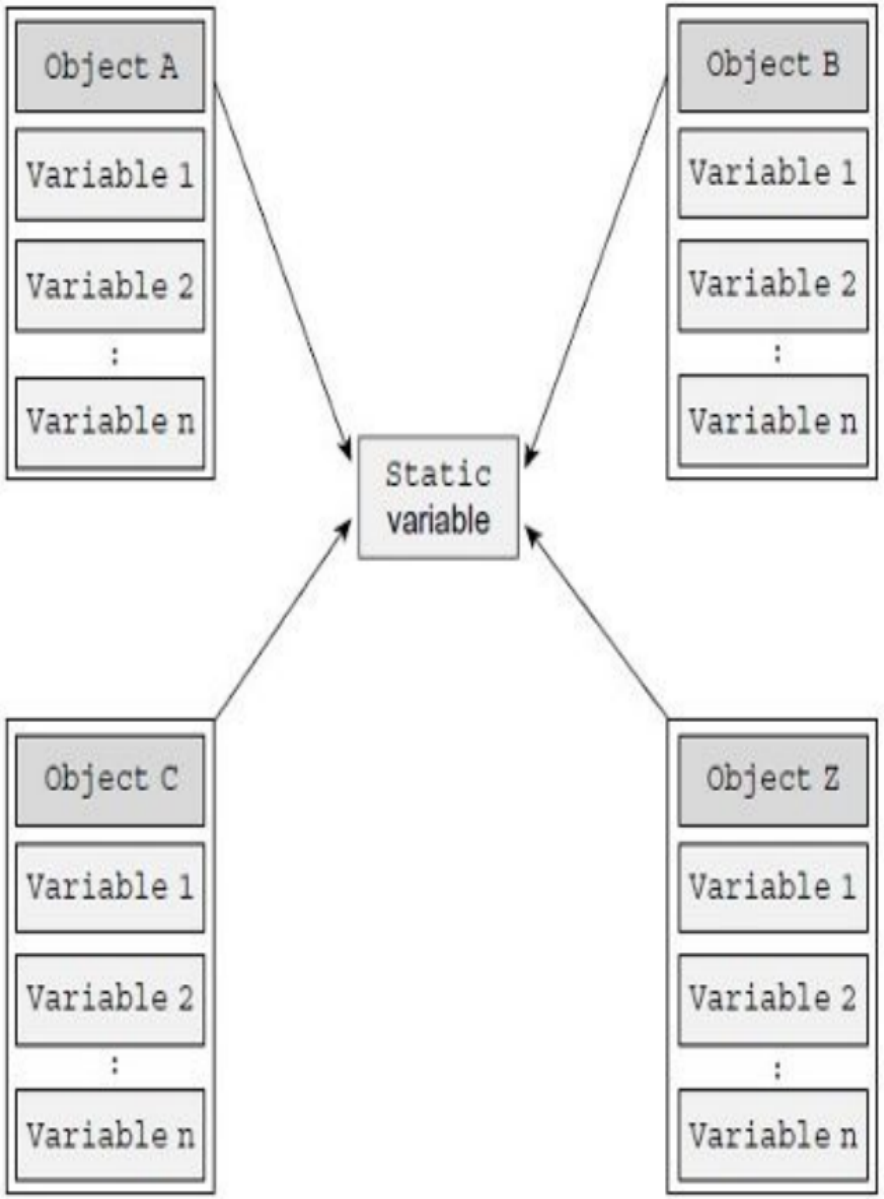
```
#include <iostream>
using namespace std;
class B;        // forward declarartion.
class A
{   int x;
    public:
    void setdata(int i)    {
        x=i;
    }
    friend void min(A,B);    // friend function
};
class B
{  int y;
    public:
    void setdata(int i)    {
        y=i;
    }
    friend void min(A,B);      // friend function
};
void min(A a,B b)
{
    if(a.x<=b.y)
    std::cout << a.x << std::endl;
    else
    std::cout << b.y << std::endl;
}

int main()
{
   A a;
   B b;
   a.setdata(10);
   b.setdata(20);
   min(a,b);
    return 0;
 }
```

STATIC DATA MEMBER:

- The properties of a static member variable are similar to that of a static variable.

- A static member variable has certain special characteristics.

1. It is initialized to zero when the first object of its class is created. No other initialization is permitted.
2. Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
3. It is visible only with in the class but its life time is the entire program.


Object A
Variable 1
Variable 2
:
Variable n
Object B
Variable 1
Variable 2
:
Variable n
Static variable
Object C
Variable 1
Variable 2
:
Variable n
Object Z
Variable 1
Variable 2
:
Variable n

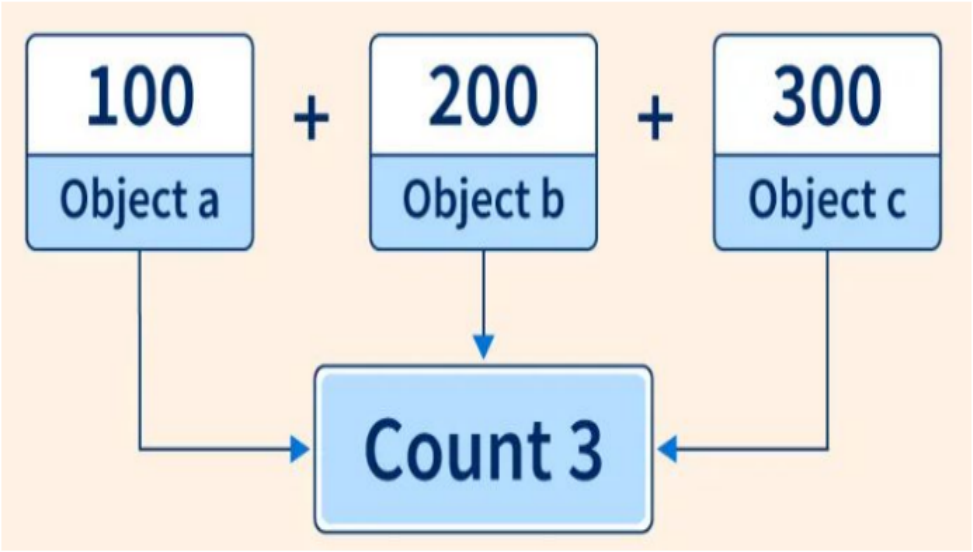
- A non-const, that is, a static data member which is not declared as const in C++, cannot be defined or initialized within the class in which it is declared.

- It is instead defined outside the class using the scope resolution operator

```
class A{
   //Declaration
   static int x;
};
//Definition
int A::x=0;
```

```
#include<iostream>
using namespace std;
class item
{
static int count; //count is static
int number;
public:
void getdata(int a)
{
    number=a;
    count++;
}
void getcount(void)
{
    cout<<"count:";
    cout<<count<<endl;
}
};
int item :: count ; //count defined
```

```
int main( )
{
item a,b,c;
a.getcount( );
b.getcount( );
c.getcount( );

a.getdata(100);
b.getdata(200);
c.getdata(300);

cout<<"after reading data : "<<endl;
a.getcount( );
b.getcount( );
c.getcount( );

return(0);
}
```



Static Member Functions

- Just like static member variables, C++ allows us to declare member functions as static.

- A static member function has the following characteristics:
- A static function can only access other static members (functions or variables) which have been declared inside the same class.
- We can only use the class name and not any of its objects to invoke a static member function.

- The syntax is as follows: class-name :: function-name();

```
#include <iostream>
using namespace std;
class item
{
   int number;
   static int count;

   public:
   void getdata(int a )
   {
      number=a;
      count++;
   }

   static void putdata( )
   {
      cout<<"count value"<<count;
   }

};
int item::count;

int main( )
{
item i1,i2;
i1.getdata(10);
i2.getdata(20);
item::putdata( );      // call static member function using class name with scope
                          resolution operator.

}
```

```
#include <iostream>
using namespace std;
class test
{
   int code;
   static int count; // static variable

   public :
   void setcode()
   {  code=++count;
   }

   void showcode(void)
   {  std::cout<<"object number" <<code <<"\n";
   }

   static void displaycount()
   {  std::cout<<"count:" <<count <<"\n";
   }
};

int test::count;
```

```
int main()
{
test t1,t2;
t1.setcode();
t2.setcode();
test::displaycount(); // accessing static function

test t3;
t3.setcode();
test::displaycount();

t1.showcode();
t2.showcode();
t3.showcode();
return 0;
}
```

```
count:2
count:3
object number1
object number2
object number3
```

OBJECTS AS FUNCTION ARGUMENTS

- Like any other data type, an object may be used as a function argument.
- Pass-by-value – A copy of object (actual object) is sent to function and assigned to the object of callee function (formal object). Both actual and formal copies of objects are stored at different memory locations. Hence, changes made in formal object are not reflected to actual object.

- Pass-by-reference – Address of object is implicitly sent to function.

- Pass-by-address – Address of the object is explicitly sent to function.

Pass-by-Value

```
#include <iostream>
using namespace std;
class Complex {
private:
    float real;
    float imag;

public:
    void read();
    void show();
    void sum(Complex c1, Complex c2);
};
void Complex::read() {
    cout << "Enter real part: ";
    cin >> real;
    cout << "Enter imaginary part: ";
    cin >> imag;
}
void Complex::show() {
    cout << "The Complex Number is: " << real << "+i" << imag << endl;
}

void Complex::sum(Complex c1, Complex c2) {
    real = c1.real + c2.real;
    imag = c1.imag + c2.imag;
}

int main() {
    Complex c1, c2, c3;

    c1.read();
    c2.read();
    c3.sum(c1, c2);
    c3.show();

    return 0;
}
```

Pass-by-Address

- Change the previous code to use Pass-by-Address

Pass-by-Address

- Change the previous code to use Pass-by-Address (explicitly)

Declaration : void sum(Complex *, Complex *);

Definition:

```
void Complex::sum(Complex *c1, Complex *c2) {
    real = c1->real + c2->real;
    imag = c1->imag + c2->imag;
}
```

Function Call: c3.sum(&c1, &c2);

Pass-by-Reference

- Change the previous code to use Pass-by-Reference

Pass-by-Reference

- Change the previous code to use Pass-by-Reference (implicitly)

Declaration : void sum(Complex &, Complex &);

Definition:

```
void Complex::sum(Complex &c1, Complex &c2) {
    real = c1.real + c2.real;
    imag = c1.imag + c2.imag;
}
```

Function Call: c3.sum(c1, c2);

Returning objects

Definition:

```
complex sum(complex c1, complex c2)
{
complex c3; // in this statement object c3 is created
c3.x=c1.x+c2.x;
c3.y=c1.y+c2.y;
return(c3); // returns object c3
}
```

Function Call: (in main)

```
complex a,b,c;
a.input(3.1,5.65);
b.input(2.75,1.2);
c=sum(a,b);
```

Constant member functions and objects

- The member functions of a class can also be declared as constant using const keyword.
- The constant functions cannot modify any data in the class.
- The const keyword is suffixed to the function prototype as well as in function definition.
- If these functions attempt to change the data, compiler will generate an error message.

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

class A {
   int c;

public:
   void add(int a, int b) const
   {       c = a + b;
           cout << "a + b = " << a + b;
   }
};

int main() {
   A a;
   a.add(5, 7);
   return 0;
}
```

Constant objects

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

class ClassA {
public:
    int x;

    void displayX() const {
        cout << "Value of x: " << x << endl;
    }
};

int main() {
    const ClassA constObject = {42};
    constObject.x = 10;
    constObject.displayX();
    return 0;
}
```

The volatile Member Function:

- In C++, one can declare a member function with volatile specifier.
- This step leads to call safely the volatile object.
- Calling volatile member function with volatile object is safe.

```
#include <iostream>
using namespace std;
class ClassA {
public:
    int x;
    void displayX() volatile
   {
        x=20;
        cout << "Value of x: " << x << endl;
    }
};

int main() {
    volatile ClassA cObject;
    cObject.x= 42;
    cObject.displayX();
    return 0;
}
```

Constant and volatile member functions

A member function declared with the const qualifier can be called for constant and nonconstant objects.

A nonconstant member function can only be called for a nonconstant object.

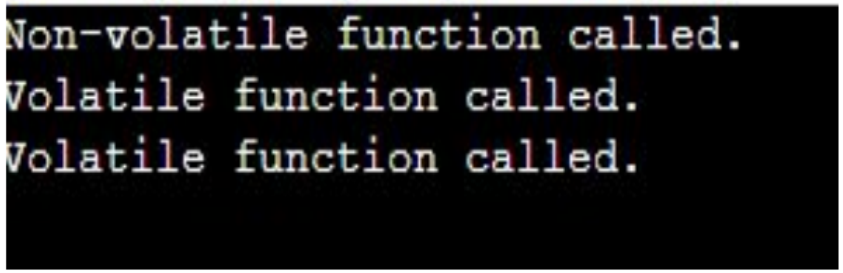
Similarly, a member function declared with the volatile qualifier can be called for volatile and nonvolatile objects.

A nonvolatile member function can only be called for a nonvolatile object.

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

class MyClass {
public:
    void nonConstFunc() {
        cout << "Non-const function called." << endl;
    }

    void constFunc() const {
        cout << "Const function called." << endl;
    }
};

int main() {
    MyClass obj;            // Non-const object
    const MyClass constObj;  // Const object

    obj.nonConstFunc();   // Allowed (non-const object calling non-const function)
    obj.constFunc();      // Allowed (non-const object calling const function)

    // constObj.nonConstFunc();  // Error: Can't call non-const function on const object
    constObj.constFunc();    // Allowed (const object calling const function)

    return 0;
}
```

```
Non-const function called.
Const function called.
Const function called.
```

```
#include <iostream>
using namespace std;
class MyClass {
public:
    void nonVolatileFunc() {
        cout << "Non-volatile function called." << endl;
    }
    void volatileFunc() volatile {
        cout << "Volatile function called." << endl;
    }
};
int main() {
MyClass obj;            // Non-volatile object
volatile MyClass volObj;  // Volatile object
obj.nonVolatileFunc();   // Allowed (non-volatile object calling non-volatile function)
obj.volatileFunc();     // Allowed (non-volatile object calling volatile function)
// volObj.nonVolatileFunc();  // Error: Can't call non-volatile function on volatile object
volObj.volatileFunc();    // Allowed (volatile object calling volatile function)
return 0;
}
```



this pointer

- C++ language uses a unique keyword called "***this***" to represent an object that invokes a member function.

- ***this*** is a pointer to the object for which "this" function was called.

Ex: The function calling A.max() set the pointer this to the address of the object A.

- The starting address is the same as the address of the first variable A in the class structure

- This unique pointer is automatically passed to a member function whenever it is called

- The pointer this acts as an implicit argument to all the member functions.

Example:

```
class ABC
{
int a;
.....
.....
};
```

private variable a can be used directly inside a member function like a=145;

we can also use this->a=123;

Example:

```
person & person :: greater(person & X)
{
if(x.age>age)
    return x; // argument object
else
    return *this; // invoking the object
}

max = A.greater(B)
```

Now implement the complete code compares the ages of two **Person** objects using the **greater()** function and **returns the object with the higher age**.

```
#include <iostream>
using namespace std;
class Person {
public:    int age;
     Person& greater(Person& x) {
          if (x.age > age)
          {          return x;  // Return the argument object
          }
          else
          {          return *this;  // Return the invoking object
          }
     }
     void display() const
     {       cout << "Age: " << age << endl;
     }
};
int main() {
Person A, B;
A.age = 25;
B.age = 30;
Person& max = A.greater(B);
cout << "The person with the greater age has: ";
max.display();
return 0;
}
```

Constructors

- A constructor is defined to be a special member function, which helps in initializing an object while it is declared.
- When a constructor is declared for a class, initialization of the class objects becomes mandatory.

Characteristics of constructors:

(1) Constructor has the same name as that of the class it belongs.

(2) It should be a public member.

(3) It is invoked implicitly at the time of creation of the objects.

(4) Constructors have neither return value nor void.

(5) The main function of constructor is to initialize objects and allocate appropriate memory to objects.

(6) Constructor can have default and can be overloaded.

(7) The constructor without arguments is called as default constructor.

Characteristics of constructors (contd.,):

(8) They are involved automatically when the object are created .

(9) They can't be inherited. But a derived class can call the base class constructor.

(10) Like other functions in C++ language, they can have default arguments.

(11) Constructors can be outside the class definition or inside the class definition.

(12) We can not refer to their address.

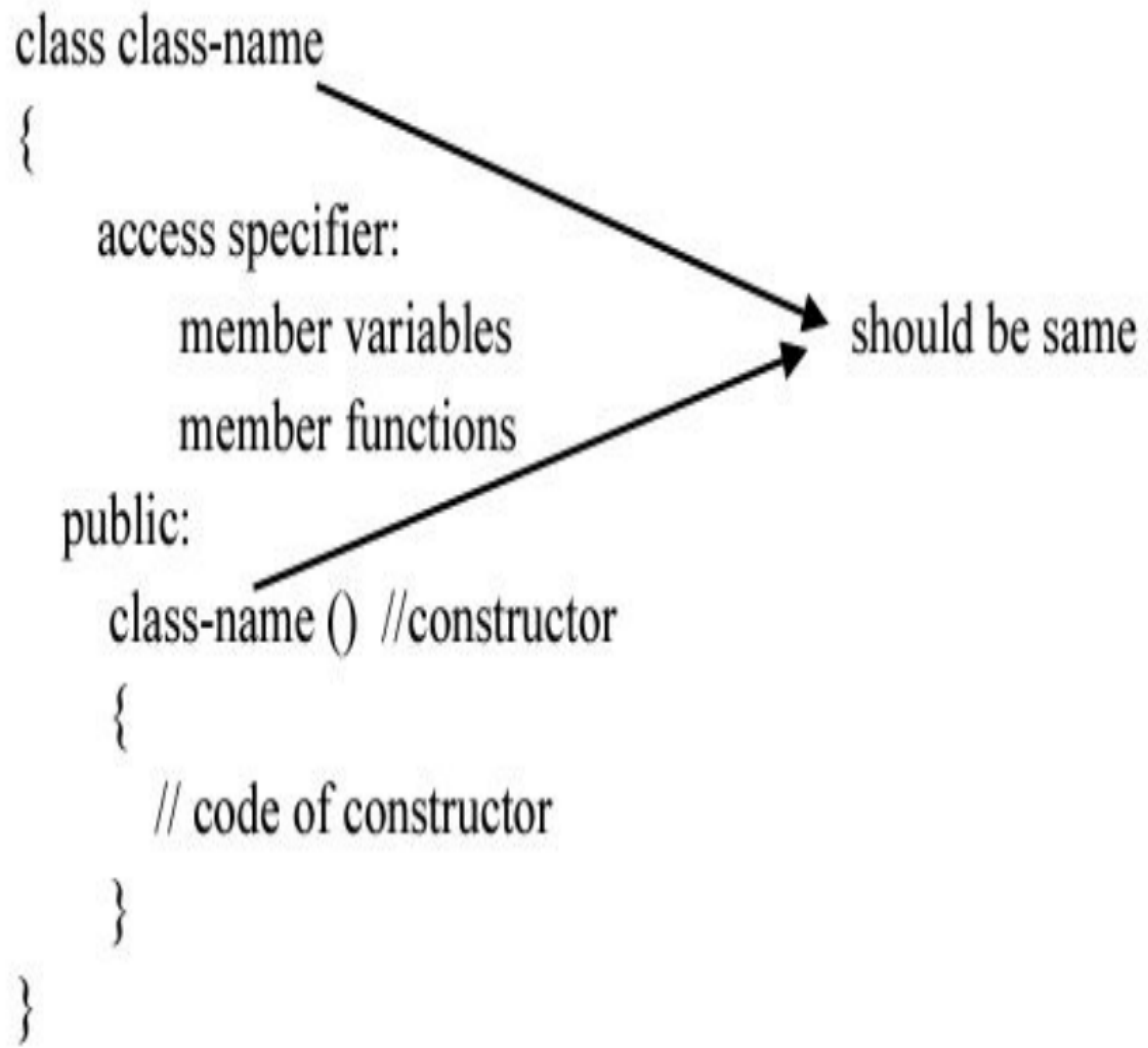
(13) Constructor can not be virtual.

(14) They make implicit calls to two operators, ***new*** and ***delete,*** when memory allocation is required.

(15) An object with a constructor (or destructor) cannot be used as a member of a union.

(16) Constructors can't be friend function.

Following is the syntax to define the constructor



class with a constructor

```
// class with a constructor
class integer
{
    int m,n;
    public:
    integer(); // declared constructor
    ……..
    …….
};
integer :: integer() // define constructor
{
    m=0;
    n=0;
}
```

Types of Constructors:

- Constructors are classified into four types:

1. Default constructor
2. Parameterized constructor
3. Copy constructor
4. Dynamic Constructor

Default Constructor:

- A constructor which does not take any arguments is called the default constructor.
- Suppose A is a class, the default constructor in the class takes the following form:

```
A()
{
    Statements;
}
```

- The statements within the body of the function assign the values to the member data of the class.

- Note that the name of the constructor is same as the class name and no return type is specified not even void.

```
#include <iostream>
using namespace std;
class construct
{
    int x;  int y;
    public:
        construct();
        void display();
};
construct::construct()
{
    x=10;
    y=20;
}
void construct::display()
{
    cout<<"The first value is:"<<x<<endl;
    cout<<"The second value is:"<<y<<endl;
}

int main()
{
    construct dc;
    dc.display();
    return 0;
}
```

Parameterized Constructor

- The constructor that can take arguments are called parameterized constructors.

```
class integer
{
int m,n;
public:
integer(int x,int y); // parameterized constructor
…….
…….
};
integer::integer(int x,int y)
{
m=x;
n=y;
}
```

Parameterized Constructor

when a constructor has been parameterized, the object declaration of ***integer class*** statement such as ,

integer int1; may not be work.

we have to pass the initial values as arguments to the constructor when an object is declared.

This can be done in **2 ways**

1. by calling the constructor **explicitly**

```
integer int1=integer(0,100); // explicitly call
```

2. by calling the constructor **implicitly**

```
integer int1(0,100); //this is implicit calling
```

Parameterized Constructor

```
#include <iostream>
using namespace std;
class construct
{
    int x;  int y;
public:
    construct(int, int);
    void display();
};
construct::construct(int val1,int val2)
{
    x=val1;
    y=val2;
}
void construct::display()
{
    cout<<"The first value is:"<<x<<endl;
    cout<<"The second value is:"<<y<<endl;
}
int main()
{
    construct pc(10,20), pc1(100,200);
    pc.display();
    pc1.display();
    return 0;
}
```

USE this POINTER

- Write a C++ program that defines a 'person' class with a 'greater' member function. The program should initialize 2 instances of the 'person' class with different names and ages. The 'greater' function should compare the ages of two 'person' objects and return a reference to the person with the greater age.

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

class person {
   char name[20];
   float age;

public:
   person(char *s, float a) {
      strcpy(name, s);
      age = a;
   }

   person &greater(person &x) {
      if (x.age >= age)
         return x;
      else
         return *this;
   }

   void display() {
      cout << "Name:" << name << "\n"
           << "Age:" << age << "\n";
   }
};

int main() {
   person p1("John", 37.50), p2("Ahmed", 29.0);
   person p = p1.greater(p2);
   cout << "Elder person is: \n";
   p.display();
   return 0;
}
```

Parameterized Constructor

- A constructor can accept a reference to its own class as a parameter.
- In such cases the constructor is called the **copy constructor**

```
Class A
{
      ……
      ……
public:
      A(A&);
};
```

What is wrong?

```
class Class1
{
private:
    int x;
public:
    Class1(int y);
};
Class1 :: Class1(int y)
{
    x = y;
}
int main()
{
    Class1 obj1(10);
    Class1 obj2;
    return 0;
}
```

Multiple Constructors in a Class

C + + permits to use more than one constructors in a single class.

Class1( ) ; // No arguments

Class1(int) ; // With arguments

Multiple Constructors in a Class

```
class add
{
    int m, n ;
  public :
    add ( ) {m = 0 ; n = 0 ;}
    add (int a, int b)
        {m = a ; n = b ;}
    add (add & i)
        {m = i.m ; n = i.n ;}
};
```

The first constructor receives no arguments.

The second constructor receives two integer arguments.

The third constructor receives one add object as an argument.

Multiple Constructors in a Class

```
class add
{
    int m, n ;
  public :
    add ( ) {m = 0 ; n = 0 ;}
    add (int a, int b)
        {m = a ; n = b ;}
    add (add & i)
        {m = i.m ; n = i.n ;}
};
```

add a1;

Would automatically invoke the first constructor and set both m and n of a1 to zero.

add a2(10,20);

Would call the second constructor which will initialize the data members m and n of a2 to 10 and 20 respectively.

Multiple Constructors in a Class

```
class add
{
    int m, n ;
  public :
    add ( ) {m = 0 ; n = 0 ;}
    add (int a, int b)
        {m = a ; n = b ;}
    add (add & i)
        {m = i.m ; n = i.n ;}
};
```

add a3(a2);

Would invoke the third constructor which copies the values of a2 into a3.

This type of constructor is called the "**copy constructor**".

Construction Overloading

More than one constructor function is defined in a class.

Multiple Constructors in a Class

```
class complex
{
    float x, y ;
  public :
    complex ( ) {  }
    complex (float a)
      { x = y = a ; }
    complex (float r, float i)
      { x = r ; y = i }
    ------
};
```

complex ( ) { }

This contains the empty body and does not do anything.

This is used to create objects without any initial values.

Multiple Constructors in a Class

In C++, every class can have a **default constructor**—a constructor that takes no parameters and is either defined

- Generated implicitly by the compiler or
- Explicitly by the user

1. Implicit Default Constructor

- The compiler automatically provides a default constructor (also known as an implicitly defined default constructor).

- This constructor does nothing special—it simply creates the object but does not perform any initialization

```
class A {
   // No constructor is explicitly defined here
};

int main() {
   A a;  // Compiler provides an implicit default constructor
   return 0;
}
```

2. User-defined Constructor

- If we define any constructor explicitly (e.g., a parameterized constructor), the compiler will not provide the default constructor automatically.

- If we still want the ability to create objects without providing explicit values during instantiation, we must also define the 'do-nothing' implicit constructor.

Multiple Constructors in a Class

- C + + compiler has an implicit constructor which creates objects, even though it was not defined in the class.
- Eg: A a; invokes the default constructor of the compiler to create the object a
- This default constructor is sometimes referred to as the "implicitly defined default constructor."
- It is a constructor that takes no arguments and performs no specific initialization.
- This works well as long as we do not use any other constructor in the class.
- However, once we define a constructor, if we still want the ability to create objects without providing explicit values during instantiation, we must also define the 'do-nothing' implicit constructor.

Defining the 'do-nothing' implicit constructor:

```
#include <iostream>
using namespace std;
class MyClass {
private:
   int x;
public:
   // Default constructor
   MyClass()  {
      // Do nothing
      x=0;
   }
   int getX() const {
      return x;
   }
};
int main() {
   MyClass obj_default;
   cout << "Default Value is " << obj_default.getX() << endl;

   return 0;
}
```

Implicitly defined default constructor by Compiler:

```
#include <iostream>
using namespace std;
class MyClass {
private:
   int x;

public:
   int getX() const {
      return x;
   }
};

int main() {
   MyClass obj_default;
   cout << "Default Value is " << obj_default.getX() << endl;

   return 0;
}
```

```
#include <iostream>
using namespace std;
class MyClass {
private:
   int x;

public:
   MyClass(int val)  {
      x=val;
   }

   // Default constructor
   MyClass()  {
      // Do nothing
      x=0;
   }

   int getX() const {
      return x;
   }
};
```

```
int main() {

   MyClass obj_with_value(42);
   cout << "Value is " << obj_with_value.getX() << endl;


   MyClass obj_default;
   cout << "Default Value is " << obj_default.getX() << endl;

   return 0;
}
```

Multiple Constructors in a class/Constructor Overloading

```
#include <iostream>
using namespace std;
class construct {
   int x;
   int y;
public:
   construct();
   construct(int);
   construct(int, int);
   void display();
};

construct::construct()
{
   x = 10;
   y = 20;
}

construct::construct(int val1)
{
   x = val1;
   y = 20;
}

construct::construct(int val1, int val2)
{
   x = val1;
   y = val2;
}

void construct::display()
{
   cout << "The first value is: " << x << endl;
   cout << "The second value is: " << y << endl;
}

int main() {
   construct obj1;
   construct obj2(100);
   construct obj3(100, 200);

   obj1.display();

   obj2.display();

   obj3.display();

   return 0;
}
```

Constructor With Default Arguments

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

class Simple{
   int data1;
   int data2;
   int data3;

   public:
      Simple(int a, int b=9, int c=8){
         data1 = a;
         data2 = b;
         data3 = c;
      }
      void printData();
};

void Simple :: printData()
{
   cout<<"The value of data1, data2 and data3 is "
            <<data1<<", "<< data2<<" and "<< data3<<endl;
}

int main(){
   Simple s1(12, 13),s2(10);
   s1.printData();
   s2.printData();
   return 0;
}
```

- Write a C++ program to find the volume of cube, sphere and cylinder using constructor overloading

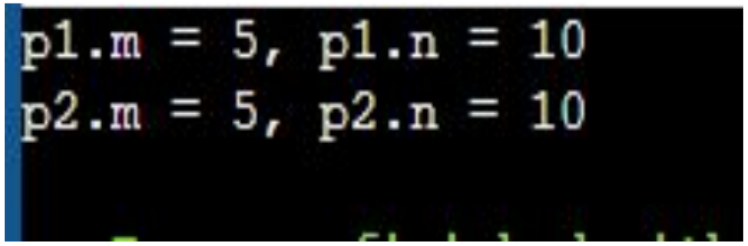
- ✔ Cube: s^3, s is the side of the cube.
- ✔ Cylinder: $\pi r^2 h$, r is the radius of circular base and h is the height of the cylinder.
- ✔ Sphere: $4/3\ \pi r^3$, r is the radius of the sphere.

```
#include <iostream>
using namespace std;
class Volume
{
    const double pi = 3.141592653589793238;  // Define pi manuallypublic:
    Volume(double side)
    {
        double volume = side * side * side;
        cout << "Volume of Cube: " << volume << endl;
    }
    Volume(double radius, double height)
    {
        double volume = pi * radius * radius * height;  // π * r^2 * h
        cout << "Volume of Cylinder: " << volume << endl;
    }
    Volume(double radius, int isSphere)
    {
        double volume = (4.0 / 3) * pi * radius * radius * radius;  // (4/3) * π * r^3
        cout << "Volume of Sphere: " << volume << endl;
    }
};
```

```
int main()
{   Volume cube(5.0);
    Volume cylinder(3.0, 7.0);
    Volume sphere(4.0, 1);
    return 0;
}
```

Copy Constructor

```
#include<iostream>
using namespace std;
class B {
private:
   int m, n;

public:
   B(){
    m=0; n=0;
   }
   B(int m1, int n1) {
      m = m1;
      n = n1;
   }
 int getm() {
      return m;
   }
   int getn() {
      return n;
   }
};

int main() {
   B p1(5, 10); // Calling Normal constructor
   B p2 = p1;   // Calling Copy constructor

   cout << "p1.m = " << p1.getm() << ", p1.n = " << p1.getn();
   cout << "\np2.m = " << p2.getm() << ", p2.n = " << p2.getn();

   return 0;
}
```



```
B p1(5, 10);
B p2 = p1;
```

OR

```
B p1(5, 10);
B p2(p1);
```

OR

```
B p1(5, 10);
B p2;
p2 = p1;
```

```
class demo1
{
   int data1, data2;
   public:
   void getdata(int a, int b)
   {
      data1=a;
      data2=b;
   }

   void showdata()
   {
      cout<<"data1= "<<data1<<"   data2= "<<data2<<endl;
   }
};
```

```
int main()
{
   demo1 obj1;
   obj1.getdata(10,20);

   cout<<"object1"<<endl;
   obj1.showdata();

   demo1 obj2=obj1;

   cout<<"object2"<<endl;
   obj2.showdata();
   return 0;
}
```

- Shallow copy is the default behavior if you don't provide a custom copy constructor.
- The compiler handles this automatically.

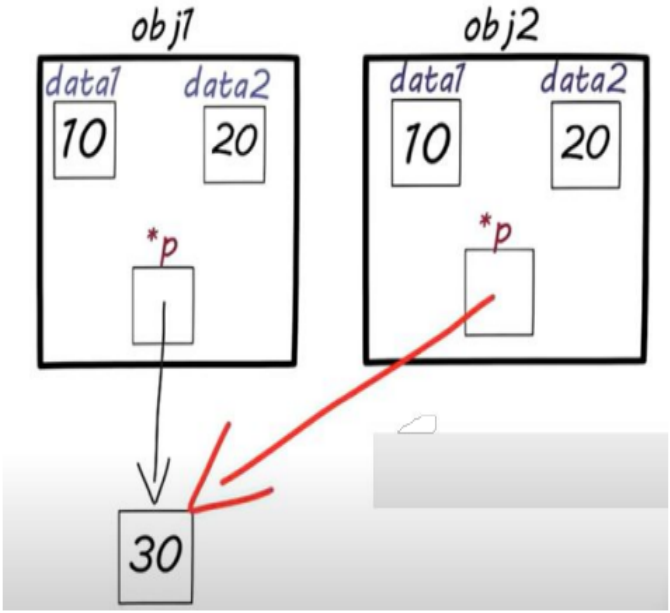
- If there is no copy constructor defined for the class, C++ uses the default copy constructor which copies each field, i.e., makes a **shallow copy**.

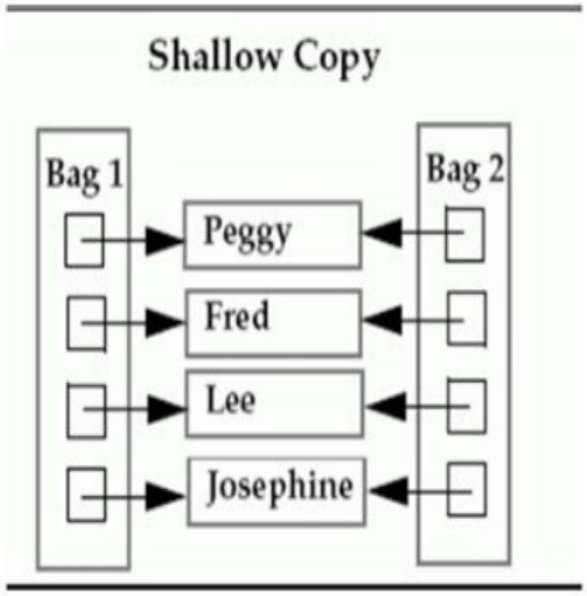
- Shallow copy is a type of copy where the contents of the copied object are directly copied to another object.

- If the object being copied contains pointers, the shallow copy only copies the addresses of the pointed data, not the data itself.

- As a result, both the original and the copied objects will point to the same dynamically allocated memory.

- *A bitwise copy of an object, where a new object is created and it has the same copy of the values in the original object, is called a Shallow copy.*





Copy Constructor:

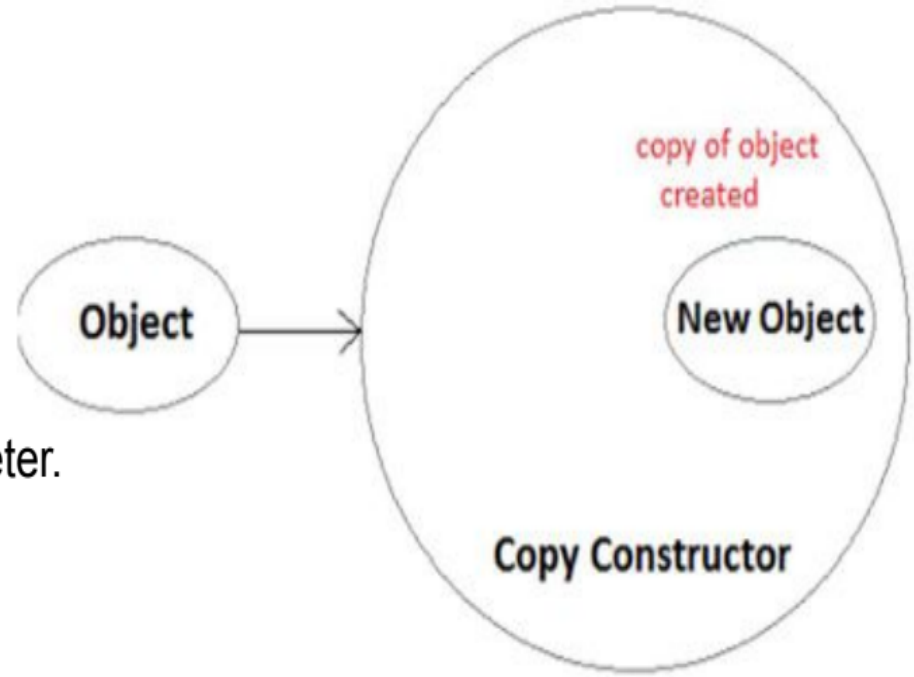
- A copy constructor is a member function which initializes an object using another object of the same class.
- It is is used to declare and initialize an object from another object of the same type.

- **Basically, the copy constructor does the following:**
- Initialize one object from another of the same type.
- Copy an object to return it from a function.
- Copy an object to pass it as an argument to a function.

Copy Constructor:

- The following is the syntax for copy constructor:

```
class-name (const class-name &obj)
{
    // constructor body
}
```



- The copy constructor takes a reference to a const parameter.
- It is const to guarantee that the copy constructor doesn't change it, and
- it is a reference because a value parameter would require making a copy, which would invoke the copy constructor.

Copy Constructor

integer (integer & i) ;

integer I2 (I1) ; or integer I2 = I1 ;

The process of initializing through a copy constructor is known as copy initialization.

The statement

I2 = I1;

will not invoke the copy constructor.

If I1 and I2 are objects, this statement is legal and assigns the values of I1 to I2, member-by-member.

Copy Constructor

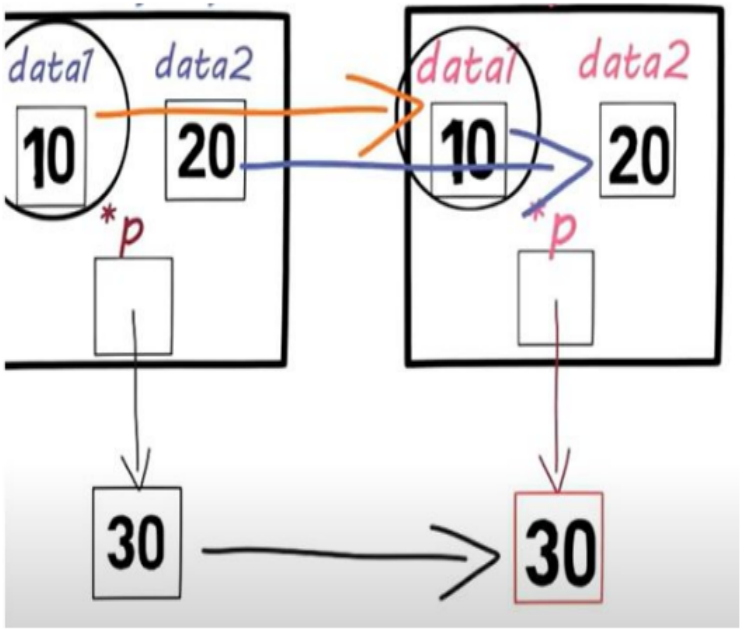
- A reference variable has been used as an argument to the copy constructor.

- We cannot pass the argument by value to a copy constructor.

Copy Constructor

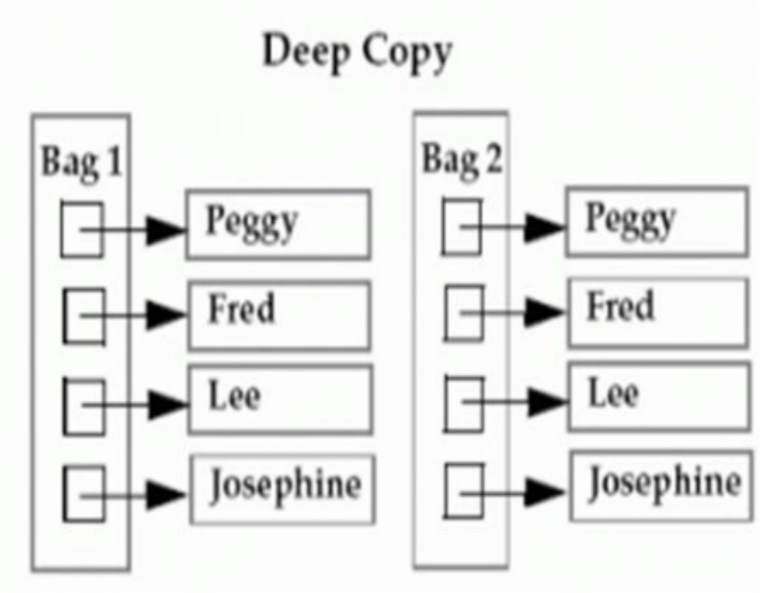
```
#include<iostream>
using namespace std;

class B {
private:
    int m, n;

public:
    B(int m1, int n1) {
        m = m1;
        n = n1;
    }

    // Copy constructor
    B(const B &p3) {
        m = p3.m;
        n = p3.n;
    }
    int getm() {
        return m;
    }

    int getn() {
        return n;
    }
};

int main() {
    B p1(5, 10); // Calling Normal constructor
    B p2 = p1;   // Calling Copy constructor

    cout << "p1.m = " << p1.getm() << ", p1.n = " << p1.getn();
    cout << "\np2.m = " << p2.getm() << ", p2.n = " << p2.getn();

    return 0;
}
```

- Deep copy, on the other hand, involves creating a new object and copying the contents of the original object, including the dynamically allocated memory pointed to by any pointers.



- Deep copy is possible only with a user-defined copy constructor.

- In a user-defined copy constructor, we make sure that pointers (or references) of copied objects point to new memory locations.



```
#include <iostream>
using namespace std;
class demo1  {
   int data1, data2, *p;
   public:
      demo1()  {
         p=new int;
      }

      demo1(demo1 &d)  {
         data1=d.data1;
         data2=d.data2;
         p=new int;
         *p=*(d.p);
       }
      void getdata(int a, int b, int c)    {
         data1=a;
         data2=b;
         *p=c;
      }
      void showdata()    {
         cout<<"data1= "<<data1<<"   data2= "<<data2<<
                     "   *p = "<< *p<<endl;
      }
};
int main()
{   demo1 obj1;
   obj1.getdata(10,20,30);
   cout<<"object1"<<endl;    obj1.showdata();
   demo1 obj2=obj1;
   cout<<"object2"<<endl;    obj2.showdata();
   return 0;
}
```

```
#include <iostream>
using namespace std;
class demo1  {
   int data1, data2, *p;
   public:
      demo1()  {
         p=new int;
      }

      demo1(demo1 &d)  {
         data1=d.data1;
         data2=d.data2;
         p=new int;
         *p=*(d.p);
          *p = 50;
       }
      void getdata(int a, int b, int c)   {
         data1=a;
         data2=b;
         *p=c;
      }
      void showdata()   {
         cout<<"data1= "<<data1<<"   data2= "<<data2<<
                     "   *p = "<< *p<<endl;
      }
};
int main()
{   demo1 obj1;
   obj1.getdata(10,20,30);
   cout<<"object1"<<endl;    obj1.showdata();
   demo1 obj2=obj1;
   cout<<"object2"<<endl;    obj2.showdata();
   return 0;
}
```

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

class MyClass {
    public:
        int *x;
        MyClass(int val) {
            x = new int(val);
        }
        MyClass(const MyClass &obj) {
            x = obj.x;
        }
};

int main() {
    MyClass obj1(10);
    MyClass obj2 = obj1;
    *obj2.x = 20;
    cout << *obj1.x << " " << *obj2.x;
    return 0;
}
```

Output: 20 20

Deep copy

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

class MyClass {
    public:
        int *x;
        MyClass(int val) {
            x = new int(val);
        }
        MyClass(const MyClass &obj) {
            x = new int(*obj.x);
        }
};

int main() {
    MyClass obj1(10);
    MyClass obj2 = obj1;
    *obj2.x = 20;
    cout << *obj1.x << " " << *obj2.x;
    return 0;
}
```

Output:10 20

```
#include<iostream>
using namespace std;
class code
{
   int id;

public:
   code ( ) { } //constructor
   code (int a) { id=a; } //constructor
   code(code &x)    {
      id=x.id;
   }

   void display( )    {
      cout<<id;
   }
};
int main( )
{
code A(100);
code B(A);
code C=A;
code D;
D=A;
cout<<" \nid of A :", A.display( );
cout<<" \nid of B :", B.display( );
cout<<" \nid of C:", C.display( );
cout<<" \nid of D:", D.display( );
}
```

Dynamic Constructors

- The constructors can also be used to allocate memory while creating objects.
- Allocation of memory to objects at the time of their construction is known as dynamic construction of objects.
- The memory is created with the help of the **new** operator.

```
#include <iostream>
using namespace std;
 class myclass {
        int* ptr;
        public:
        myclass()  {
                ptr = new int;
                *ptr = 10;
        }

        void display()
        {
                cout<< *ptr<<endl;
        }
};
```

```
int main()
{
        myclass obj1;
        obj1.display();
        return 0;
}
```

Destructors

A destructor is used to destroy the objects that have been created by a constructor.

- It should be public member

Like constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.

eg: ~ integer ( ) { }

Destructors

A destructor never takes any argument nor does it return any value.

It will be invoked implicitly by the compiler upon exit from the program – or block or function as the case may be – to clean up storage that is no longer accessible.

It is a good practice to declare destructors in a program since it releases memory space for further use.

Whenever new is used to allocate memory in the constructor, we should use delete to free that memory.

Destructors

```
#include <iostream>
using namespace std;
int count=0;
class integer
{    int x;
     public:
     integer(int y)
     {            count++;
          cout<<"object "<<count<<" created"<<endl;
          x=y;
     }
     ~integer()
     {            cout<<"object "<<count<<" destroyed"<<endl;
          count--;
     }
};
int main()
{     integer a(10),b(20);
      return 0;     }
```

Guess the output

```
#include <iostream>
using namespace std;
class Engine
{
    public:
    Engine()
    {     cout << "Engine constructor called." << endl;    }
    ~Engine()
    {      cout << "Engine destructor called." << endl;    }
};
class Car
{
    Engine engine;  // Engine object as a member of Car class
     public:
    Car()
    {      cout << "Car constructor called." << endl;    }
    ~Car()
    {      cout << "Car destructor called." << endl;    }
};
```

```
int main()
{
cout << "Creating Car object..." << endl;
Car myCar;  // Create Car object, which contains an Engine object
cout << "Car object created." << endl;
return 0;
}
```

OUTPUT:

```
Creating Car object...
Engine constructor called.
Car constructor called.
Car object created.
Car destructor called.
Engine destructor called.
```

Copy constructor vs Assignment Operator

Copy constructor is called when a new object is created from an existing object, as a copy of the existing object.

Assignment operator is called when an already initialized object is assigned a new value from another existing object.

```
MyClass t1, t2;     //constructor
MyClass t3 = t1;  // calls copy constructor
t2 = t1;                // calls assignment operator.
```

Copy constructor vs Assignment Operator

```
class Point {
        public:
              . . .
       Point(const Point &p);   // copy constructor
       . . . } ;
Point::Point(const Point &p)  {
       x = p.x;
       y = p.y;
}
int main() {
       Point p;          // calls default constructor
       Point s = p;      // calls copy constructor
       p = s;            // assignment, not copy constructor
return 0; }
```