Constructors And Destructors

Constructors

- A special member function whose task is to initialize the objects of its class.
- Its name is same as the class name.
- The constructor is invoked whenever an object of its associated class is created.
- Eg:

```
class sample
{
  int m,n;
  public:
  sample(void); // Constructor declared
  ………….
};
sample::sample(void) // Constructor defined
{
  m=0;n=0;
}
```

sample ob1;

Constructors

```
#include <iostream>
using namespace std;
class Counter
{
private:
unsigned int count;
public:
Counter()  //constructor
{
 count=0;
}
void inc_count() //increment count
{
count++;
}
int get_count() //return count
{
return count;
}
};
```

```
int main()
{
Counter c1, c2; //define and initialize
cout << “\nc1=” << c1.get_count(); //display
cout << “\nc2=” << c2.get_count();
c1.inc_count(); //increment c1
c2.inc_count(); //increment c2
c2.inc_count(); //increment c2
cout << “\nc1=” << c1.get_count(); //display again
cout << “\nc2=” << c2.get_count();
}
```

Initializer List

- In the Counter class the constructor is initializes the count member to 0, like this:

```
Counter()
{
  count = 0;
}
```

- Another way to initialize a data member:

```
Counter() : count(0)
{ }
```

- In case of multiple member initializations, they must be separated by commas. The result is the *initializer list (sometimes called by other names, such as the member-initialization list)*.

```
someClass() : m1(7), m2(33), m3(4)
{ }
```

Default Constructor

- Constructor with no argument is called 'default Constructor'.
- If compilers declares the 'default constructor', then it is said to be '**implicitly declared default constructor**‘, otherwise it is said to be a “**explicitly declared default constructor” or** “**u**ser define no argument constructor”

Parameterized Constructor

- Constructors that can take arguments are called parameterized constructors.
- Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.
- The constructor sample can be modified to take arguments as shown

```
class sample
{
   int m,n;
   public:
            sample(int x, inty)
             {m=x;
              n=y;
             }
};
```

Parameter passing for parameterized constructor

We pass the initial values as arguments to the constructor function when an object is declared.

Eg:

```
Sample s1(20,30); //implicit call
```

or

```
Sample s2=sample(20,30); //explicit call
```

Default Copy Constructor

- A copy constructor is used to declare and initialize an object from another object.
- It’s a one argument constructor whose argument is an reference to object of the same class as the constructor
- Eg:

  ```
  sample s3(s2);
  ```

  - Defines the object s3 and at the same time initializes it to the values of object s2
  - Another form of the statement is

  ```
  sample s3=s2;
  ```

Default Copy Constructor

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

class Distance              //Distance class
{
        private:   int feet; float inches;
        public:
Distance() : feet(0), inches(0.0)         //constructor (no args)
{ }
Distance(int ft, float in) : feet(ft), inches(in)   //constructor (two args)
{ }

void showdist()                    //display distance
{       cout << feet << “\’-” << inches << ‘\”’; }
};
```

//Note: no one-argument constructor is declared

Default Copy Constructor

```
int main()
{
        Distance dist1(11, 6.25);            //two-arg constructor
        Distance dist2(dist1);         //one-arg constructor
        Distance dist3 = dist1;              //also one-arg constructor
        //display all lengths
        cout << “\ndist1 = “; dist1.showdist();
        cout << “\ndist2 = “; dist2.showdist();
        cout << “\ndist3 = “; dist3.showdist();
        cout << endl;
return 0;
}
```

```
O/p
dist1 = 11’-6.25”
dist2 = 11’-6.25”
dist3 = 11’-6.25”
```

- Distance dist2(dist1); This causes the default copy constructor for the Distance class to perform a member-by-member copy of dist1 into dist2.
- Surprisingly, A different format has exactly the same effect, causing dist1 to be copied member-by-member into dist3:
- **Distance dist3 = dist1;**
  Although this looks like an assignment statement, it is not.
- Both formats invoke the default copy constructor, and can be used interchangeably.

Multiple constructor in a class (Constructor overloading)

```
class sample
{
 int m,n;
  public:
  sample(){m=0;n=0;};
  sample(int x,int y){m=x;n=y;}
  sample(sample &i){m=i.m;n=i.n;}        // also called
  copy constructor
};
```

Objects created as follows

```
sample  s1;              // invokes first constructor
sample s2(10,10);                // invokes second
  constructor
sample s3(s2);               // invokes third constructor
```

Constructor with Default Arguments

- Just like other member function constructor also can be defined with default arguments.

```
class add
{  private:     int num1, num2,sum;
    public:     add(int=0,int=0);         //Default argument constructor to reduce
};                      //the number of constructors
add::add(int n1, int n2)
{       num1=n1;
        num2=n2;
        sum=num1+num2;
        cout<<“num1+num2=”<<sum<<endl;
}
int main()
{
        add obj1, obj2(5), obj3(10,20);
        return 0;
}
```

```
O/p
num1+num2 =0
num1+num2 =5
num1+num2 =30
```

Constructor with Default Arguments(cont…)

```
class add
{
private:   int num1, num2,sum;
public:     add(int=0,int=0);          //Default argument constructor
        add(){}   //Default constructor
};
add::add(int n1, int n2)
{
        num1=n1;
        num2=n2;
        sum=num1+num2;
        cout<<“num1+num2=”<<sum<<endl;
}
int main()
{
        add obj1, obj2(5), obj3(10,20);
        return 0;
}
```

O/p
Syntax Error:Call of Overloaded ‘add()’ is ambiguous

Important Points About Constructor

Q1:What happens when we write only a copy constructor – does compiler create default constructor?

Compiler doesn’t create a default constructor if we write any constructor.

If user have not provided any of the following constructor, then the compiler declares the default constructor for you:

a)Copy Constructor (User defined copy constructor)

b)Non-default constructor(Parameterized constructor)

c)default constructor (user define no argument constructor)

Q2:what happens when we write a normal constructor and don’t write a copy constructor?

Compiler creates a copy constructor if we don’t write our own.

Compiler creates it even if we have written other constructors in class.

Properties of Constructors

1. **Same Name as the Class: This** is one way the compiler knows they are constructors.
2. **No return type is used:** Since the constructor is called automatically by the system, there’s no program for it to return anything to; a return value wouldn’t make sense. This is the second way the compiler knows they are constructors.
3. These are called automatically when the objects are created.
4. These should be declared in the public section for availability to all the functions.
5. These cannot be inherited, but a **derived class can call the base class constructor.**

These cannot be static.

Properties of Constructors(Cont…)

7. Default and copy constructors are generated by the compiler wherever required.
8. These can have default arguments as other C++ functions.
9. A constructor can call member functions of its class.
10. An object of a class with a constructor cannot be used as a member of a **union** **(Why?).**
11. Constructor make **implicit calls to the memory allocation operator new.**
12. These cannot be **virtual.**

What Will be the Output?

```
#include <iostream>
using namespace std;
class Point{
        int x, y;
        public:
        Point(const Point &p)
        {
                x = p.x;
                y = p.y;
                cout<<“User Defined Copy constructor”;
        }
};
int main(){

        Point p1;
        Point p2 = p1;
        return 0;
}
```

// COMPILER ERROR: No matching function for call to Point::Point();

Destructors

- Destructor is used to destroy the object created by constructor.
- Destructor has same name as the class name but preceded by a tilde(~) symbol.
- Destructor is also a member function.
- Destructor does not take any arguments and also don’t return any value.
- A destructor is invoked (called) when an object of the class goes out of scope, or when the memory space used by it is de allocated with the help of **delete operator.**
- **Declaration and Definition of a Destructor**

The syntax for declaring a destructor is :

~name_of_the_class()

{

Program to illustrate the execution of destructor

Class definition

```
#include<iostream>
int count=0;
class test
{
 public:
 test(){ count++;
      cout<<“ object”<<count<<“Created”;
     }
 ~test(){
     cout<<“object”<<count<<“Destroyed”;
     count--;
}
};
```

main() function

```
int main()
{
cout<<“\n Enter Main”;
test t1,t2;
{
 cout<<“Enter block 1\n”
 test  t3;
}
{
 cout<<“\n Enter Block 2\n”;
  test t4;
}
cout<<“\n Reenter main”;
return 0;
}
```

Output of the program

```
Enter Main
Object 1 Created
Object 2 Created
Enter Block 1
Object 3 Created
Object 3 Destroyed
Enter Block 2
Object 3 Created
Object 3 Destroyed
Reenter main
Object 2 Destroyed
Object 1 Destroyed
```

New and delete Operators

- **Dynamic Memory Allocation/ Deallocation Operators Using new, delete:-**
- The syntax of the new operator is given below :

  pointer_variable = **new data_type;**
- Where the data type is any allowed C++ data type and the pointer_variable is a pointer of the same data type. For example,
- char * cptr ; cptr = new char;
- The above statements allocate 1 byte and assigns the address to cptr.
- The following statement allocates 21 bytes of memory and assigns the starting address to cptr :

New and delete Operators

- We can also allocate and initialize the memory in the following way :
- Pointer_variable = **new data_type (value);**
- Where value is the value to be stored in the newly allocated memory space and it must also be of the type of specified data_type. For example,
- char *cptr = new char (‘j’);
- int *empno = new int (size); //size must be specified

delete Operator

- It is used to release or deallocate memory. The syntax of **delete operator is :**
- delete_pointer_variable;
- For example,

delete cptr;

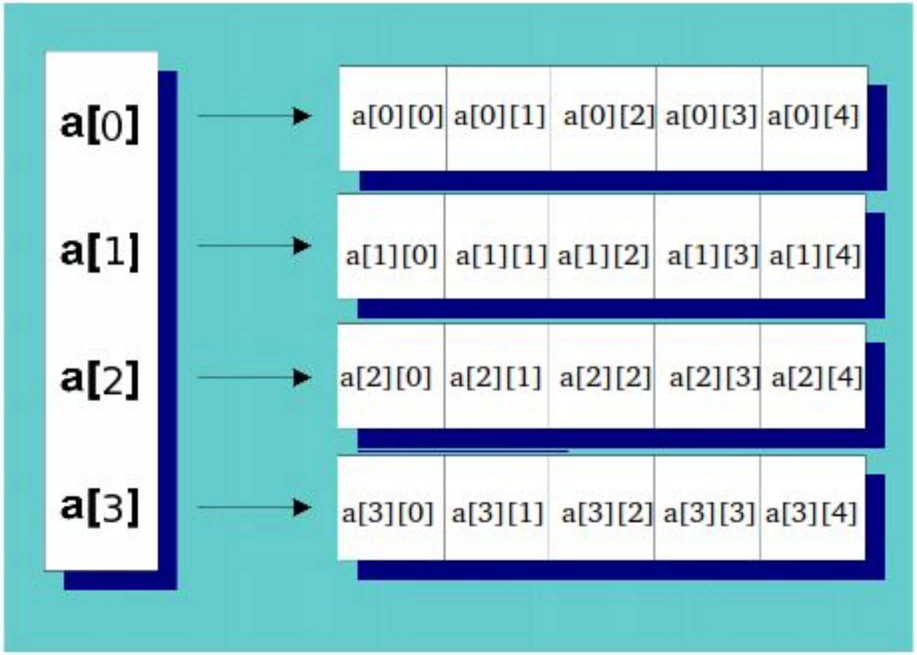
delete [ ] empno; //some versions of C++ may require size

Declaring 2 D array using new

- A dynamic 2D array is basically an array of *pointers to arrays*.
- Need to Initialize using a loop, like this:

```
  int** ary = new int*[rowCount];
 for(int i = 0; i < rowCount; ++i)
             ary[i] = new int[colCount];
```

The above, for colCount= 5 and rowCount = 4, would produce the following:


a[0]
a[1]
a[2]
a[3]
a[0][0] a[0][1] a[0][2] a[0][3] a[0][4]
a[1][0] a[1][1] a[1][2] a[1][3] a[1][4]
a[2][0] a[2][1] a[2][2] a[2][3] a[2][4]
a[3][0] a[3][1] a[3][2] a[3][3] a[3][4]

and then clean up would be:

```
for(int i = 0; i < rowCount; ++i)
   {
         delete [] ary[i];
   }
 delete [] ary;
```

Dynamic constructors

- Dynamic constructor can be used to allocate the right amount of memory for each object when the object are not of the same size, this result in saving of memory.
- Provides flexibility of using different format of data at runtime depending upon the situation.
- Allocation of memory to objects at the time of their construction is known as dynamic construction of objects.
- The memory is allocated with the help of the *new* operator.

Program to illustrate dynamic constructors

Program to concatenate the strings and display

```
#include<iostream>
#include<string.h>
class String
{
     char *name;
    int   length;
    public:
    String(){ } //default constructor
    String(char *s){
            length=strlen(s);
            name=new char[length+1];
            strcpy(name,s);
            }
    void display(void){cout<<name<<"\n";
                }
    void join(String &a,String &b);
};
```

join function definition

```
void String::join(String &a,String &b)
{
    length=a.length+b.length;
   name=new char[length+1];
   strcpy(name,a.name);
   strcat(name,b.name);
}
```

main() function

```
int main()
{
    char *first=“Joseph”;
   String
   name1(first),name2(“john”),name3(“jack”),s1,s2;
   s1.join(name1,name2);
   s2.join(s1,name3);
   name1.display();
   name2.display();
   name3.display();
   s1.display();
   s2.display();
   return 0;
}
```

```
    Joseph
john
jack
joseph john
joseph john jack
```

THIS POINTER

- The ‘this’ pointer is passed as a hidden argument to all **nonstatic** member function calls and is available as a local variable within the body of all nonstatic functions.
- ‘this’ pointer is a constant pointer that holds the memory address of the current object.
- For example when you call obj.func(),
- ‘**this**’ will be set to the address of **obj**.
- For a class X, the type of this pointer is ‘**X* const**’.
- Also, if a member function of X is declared as const, then the type of this pointer is ‘**const X *const**’

Following are the situations where ‘this’ pointer is used:

```
#include<iostream>
using namespace std;
class Test{
  int x;
  public:
  void setX (int x){/* local variable is same as a
  member's name */          this->x = x;  //This
  pointer is used
       }
  void print() { cout << "x = " << x << endl;
       }
};
```

main function

```
int main()
{
   Test obj;
   int x = 20;
   obj.setX(x);
   obj.print();
   return 0;
}
```

To return reference to the calling object

- /* Reference to the calling object can be returned */

```
Test& Test::func ()
{
   // Some processing
   return *this;
}
```

Practice Programs

- Create class account with data members name, accno, balance, branch. Create 1 object with 4 inputs for account class & display the same.
- WAP to add two complex numbers using constructors with default argument.