INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

EEC-101
Programming with C++

Module-4:
Object Oriented Programming

About Subject

- Object Oriented Programming Concepts
  - Data hiding,
  - Abstract data types,
  - Classes and Access control;
  - Class Implementation-
    - constructors, default constructor, copy constructor,
    - destructor
  - Operator overloading
  - Friend functions
  - Introduction to Templates

- C++ defines several kinds of functions that can be declared only as class members
  — these are called "special member functions."

- These functions affect the way objects of a given class are created, destroyed, copied, and converted into objects of other types.

- Another important property of many of these functions is that they can be called implicitly (by the compiler).

Constructors and Destructors

- There are two Special members functions
    1. *Constructors*
    2. *Destructors*

- Constructors are called every time we create an object.

- Destructors are called every time we destroy an object.

Constructors

- A constructor is a special member function whose task is to initialize the objects of its class. This is known as the automatic initialization of objects.
- It is special because its **name is the same as the class name.**
- The constructor is invoked whenever an object of its associated class is created.
- It is called a constructor because it constructs the values of data members of the class.

Constructor Types

- Default Constructors
- Parameterized Constructors
- Copy Constructors

Why constructors

- First a variable declared.

  Example:

  ```
  int x;  //variable x of type integer
  float y; //variable y of type float
  ```

- Variable x or y would have garbage value before initialization.

- We needed to initialize the variable to 0 or to some other useful value before using it.
  - Example:

  ```
  int x {0};  //variable x of type integer initialized to 0
  float y {22.5} ; //variable y of type float initialized to 22.5
  ```

Why Constructors

- The same is true of objects. i.e. objects need to be declared and initialized.
- The difference is that with an object, we can't just assign it a value. We can't say:

  Student s1 = 0; because that doesn't make sense. (here s1 is an object of class Student)
- A Student is not a number, so we can't just set it to 0.
- The way object initialization happens in C++ is that a special function, the constructor, is called when you instantiate an object.

Implicit Constructor

```
/** classes example with implicit constructor*/
#include <iostream>
using namespace std;
class CRectangle {

    int x, y;
    public:
    void set_values (int, int);
    int area ()
    {return (x*y);}
};
void CRectangle::set_values (int a, int b) {
x = a;
y = b;}
int main() {
CRectangle rect;
cout<<"area: " << rect.area() <<endl;// garbage
rect.set_values (4,6);
cout<<"area: " << rect.area() <<endl;
system("pause");
return 0;
}
```

```
area: 0
area: 24
Press any key to continue . . .
```

```
/*** example: one class, two objects ***/
#include <iostream>
using namespace std;
class CRectangle {
    int x, y;
    public:
    void set_values (int, int);
    int area () { return (x*y);}
};
void CRectangle::set_values (int a, int b) {
x = a;
y = b;}
int main() {
CRectangle small_rect, big_rect;
small_rect.set_values (4,6);
big_rect.set_values (7,5);
cout<<"Small rectangle area: " << small_rect.area() << endl;
cout<<"Big rectangle area: " << big_rect.area() << endl;
system("pause");
return 0;
}
```

```
Small rectangle area: 24
Big rectangle area: 35
Press any key to continue . . .
```

Parameterized Constructor

```
/** example: class constructor**/
#include <iostream>
using namespace std;
class CRectangle {
int width, height;
public:
CRectangle (int, int); //constructor prototype
int area () { return (width*height); }
};
// constructor definition (name is same as class name)
CRectangle::CRectangle (int a, int b)
{
width= a;
height = b;
}
int main() {
CRectangle small_rect (4,6);
CRectangle big_rect (7,5);
cout<<"Small rectangle area: " << small_rect.area() << endl;
cout<<"Big rectangle area: " << big_rect.area() << endl;
system("pause");
return 0;
}
```

```
Small rectangle area: 24
Big rectangle area: 35
Press any key to continue . . .
```

Without constructor

```
small_rect.set_values (4,6);
big_rect.set_values (7,5);
```

With constructor

```
CRectangle small_rect (4,6);
CRectangle big_rect(7,5);
```

(Syntax Constructor Definition)

Class_name::*Class_name*(*parameter*$_1$, ..., *parameter*$_n$)

{

statements

}

Example:

```
Point::Point(double xval, double yval)
{
  x = xval; y = yval;
}
```

```
CRectangle::CRectangle (int a, int b)
{
width= a;
height = b;
}
```

- Constructors can not be called explicitly as if they were regular member functions.
- They are only executed when a new object of that class is created.
- Neither the constructor prototype declaration (within the class) nor the latter constructor definition include a return value; not even void.

Overloading Constructors

- Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters.

- For overloaded functions the compiler will call the one whose parameters match the arguments used in the function call.
- In the case of constructors, which are automatically called when an object is create

- The one executed is the one that matches the arguments passed on the object declaration.

Example: Constructor Overloading

```
/**overloading class constructors*/
#include <iostream>
using namespace std;
class CRectangle {
    int width, height;
public:
    CRectangle(); //default constructor
    CRectangle (int, int); // parametrized constructor
    int area (void){
    return (width*height);}
    };
CRectangle:: CRectangle () {
width = 7;
height = 7;
}
CRectangle::CRectangle (int a, int b) {
width= a;
height = b;
}
int main() {
CRectangle small_rect (4,6);
CRectangle big_rect; //default constructor is invoked
cout << "Small rectangle area: " << small_rect.area() << endl;
cout << "Big rectangle area: " << big_rect.area() << endl;
system("pause");
}
```

```
Small rectangle area: 24
Big rectangle area: 49
Press any key to continue . . .
```

```
/**overloading class constructors*/
#include <iostream>
using namespace std;
class CRectangle {
    int width, height;
public:
        CRectangle (int, int); // parametrized constructor
    int area (void){
    return (width*height);}
    };
CRectangle::CRectangle (int a, int b) {
width= a;
height = b;
}
int main() {
CRectangle small_rect (4,6);
CRectangle big_rect; //default constructor is invoked
// this leads to error no matching function
cout << "Small rectangle area: " << small_rect.area() << endl;
cout << "Big rectangle area: " << big_rect.area() << endl;
system("pause");
}
```

Default Constructor

- A constructor that has zero parameter list or in other sense, a constructor that accepts no arguments is called a zero-argument constructor or default constructor.

- The default constructor for class CRetangle is

```
CRectangle::CRectangle ()
{ width = 0; height = 0; }
```

- If no such constructor is defined, then only the compiler supplies the default constructor. Therefore a statement such as
- Crectangle small_rect; invokes the default constructor of the compiler to create the object small_rect

Using Default Constructor with parameters

- If a new object is declared and wants to use its default constructor (the one without parameters), we do not include parentheses ():

- CRectangle rectb; *// right*
- CRectangle rectb(); *// wrong!*

Constructor with Default Arguments

```
//Only one default constructor per class
#include <iostream>
using namespace std;
class CRectangle {

int width, height;
public:
//CRectangle (); //default constructor (gives error because parameterized constructor can act like default constructor
CRectangle (int=1,int=1 ); // parametrized constructor with default arguments becomes default constructor
int area (void) {return (width*height);}
};

CRectangle::CRectangle (int a, int b) //parameterized constructor (no return value)
{
width= a;
height=b;}
int main() {
CRectangle small_rect (3,3);//parameterized constructor invokes
CRectangle medium_rect(2);//parameterized constructor with default arguments
CRectangle big_rect; //default constructor is invoked
cout << "Small rectangle area:" << small_rect.area() << endl;
cout<<"Medium rectangle area:  " << medium_rect.area() << endl;
cout<<"Big rectangle area: " <<big_rect.area() << endl;
system("pause"); return 0;
}
```

```
Small rectangle area:9
Medium rectangle area:  2
Big rectangle area: 1
Press any key to continue . . .
```

Characteristics of Constructors

- They should be declared mostly in public section
- Invoked automatically when objects are created
- Do not have return type, not even void. They can not return values.

Summary

- An constructor is called every time an object is instantiated
- Uses of constructor
  - Initialize data members when required
    (data members can not be initialized in the class declaration because each object has its own variables and must be initialized with the values required by the object being created)
  - To validate data and thus make the program more robust
  - To allocate memory for an object

Summary

- Name of the function is same as the class name. No return type, not even void
- A class must have at least one constructor, either defined by the program or by the compiler
- Default constructor is called whenever an object is created without passing any arguments
- **If we define any type of constructor, we must also define a default constructor, if it is needed**

If we do not specify any constructor

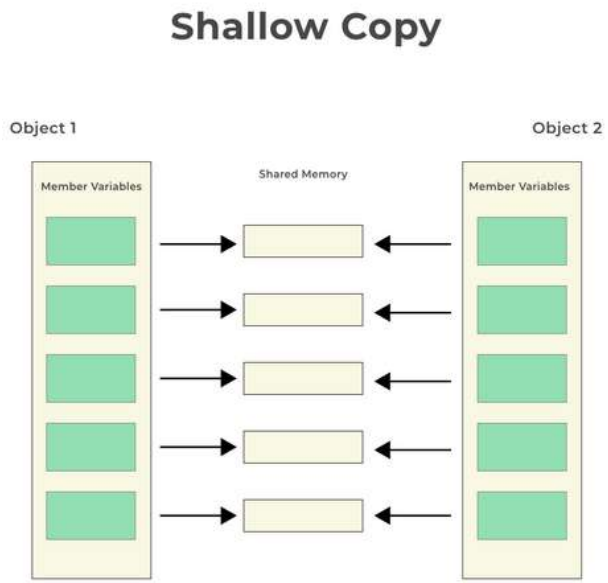
- Three special member functions in total that are implicitly declared if we do not declare our own constructor (explicitly)
- These are
  - the default destructor
  - the *copy constructor*,
  - the *copy assignment operator*

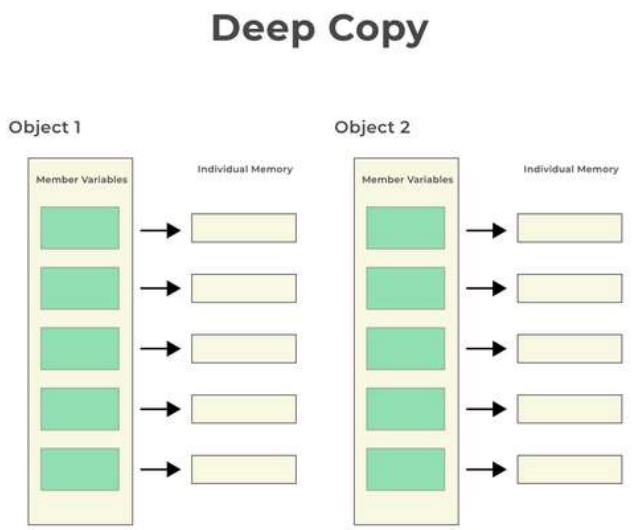
Copy constructor

- Copy constructor is a constructor function with the same name as the class used to make deep copies of objects.
- There are 3 important places where a copy constructor is called.
  1. When an object is created from another object of the same type
  2. When an object is passed by value as a parameter to a function
  3. When an object is returned from a function (unless return value optimization applies- the return value optimization (RVO) is a compiler optimization that involves eliminating the temporary object created to hold a function's return value. This will depend upon the compiler)

Shallow Copy vs Deep Copy

The main difference between a shallow copy and a deep copy is that a shallow copy creates a new object that references the original data, while a deep copy creates an independent copy of the original object.





Shallow copy vs Deep copy

- Shallow copy creates a new object and inserts references to the objects in the original.
- For example, a shallow copy of an array creates a new array that points to the same elements as the original array. Changes made to the original object will also be reflected in the copied object. Shallow copies can be used to optimize performance and share data structures between objects.
- Deep copy creates a new object and inserts copies of the objects in the original. For example, a deep copy of an array creates a completely independent copy of the array and its data. The original and copied objects do not share any data, so changes made to one will not affect the other.

Default copy constructor

- If a copy constructor is not defined in a class, the compiler itself defines one. This will ensure a shallow copy.

- If the class has pointer variables with dynamically allocated memory, then one needs to worry about defining a copy constructor. It can not be left to the compiler's discretion.

- If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.

- For ex:

```
class A   //Without copy constructor
{
    private:
    int x;
    public:
    A() {x = 10;} //default constructor
    ~A() {}         // destructor
 };
```

```
class B    //With copy constructor
        {
            private:
            char *name;
            public:
            B()
            {
            name = new char[20];
            }
            ~B()
            {
            delete [] name;
            }
          //Copy constructor
            B(const B  &b)
            {
            name = new char[20];
            strcpy(name, b.name);
            }
         };
```

- Let us imagine if you don't have a copy constructor for the class B. At the first place, if an object is created from some existing object, we cannot be sure that the memory is allocated. Also, if the memory is deleted in destructor, the delete operator might be called twice for the same memory location.
- This is a major risk. One happy thing is, if the class is not so complex this will come to the fore during development itself. But if the class is very complicated, then these kind of errors will be difficult to track.
- In Windows this will lead to an application popup and UNIX will issue a core dump. A careful handling of this will avoid a lot of nuisance.

- const(Constant) objects and const Member functions
- Keyword const specify that an object is not modifiable and that any object to modify the object will result in compilation error
- C++ disallows member functions calls for const objects unless the member functions themselves are also declared const
- A constructor must be a non-const member function, but can still be used to initialize a const object
- “constness” of the const object is enforced from the time the constructor completes initialization of the object until object that object’s destructor is called

Example

```
//class called Point, with all necessary special member functions
#include<iostream>
#include<cmath>
using namespace std;
class point
{
    float x,y,z;
    public:
    point () //special member function default constructor
    { x=y=z=0; }
    point (float i,float j,float k)//special member function parameterized constructor
    {x=i;y=j;z=k;}
    point (point &a)//special member function- copy constructor
    {x=a.x; y=a.y; z=a.z;}
    float Negate()//member function
    { x=-x;y=-y;z=-z;}
    void print()//member function
    { cout<<" ("<<x<<", "<<y<<", "<<z<<") "<<endl;}
    float norm_dist ()//member function
    {return (sqrt(x*x+y*y+z*z));}
};
```

```
The point has the coordinates
 (2, 3, 4)
The point coordinates after negation
 (-2, -3, -4)
Normal Distance of the point from (0,0,0) is 5.38516
The coordinates of the point pl after copy constructor is  (2, 3, 4)

Press any key to continue . . .
```

```
int main(){
point p(2,3,4);// initialized using parameterized constructor
point pl(p);// p is copied to pl using copy constructor
cout<<"The point has the coordinates"<<endl;
p.print();
cout<<"The point coordinates after negation"<<endl;
p.Negate();
p.print();
cout<<"Normal Distance of the point from (0,0,0) is ";
cout<<p.norm_dist () <<endl;
cout<<"The coordinates of the point pl after copy constructor is ";
pl.print();
cout<<endl;
system("pause");
return 0;}
```

Destructors

- Destructors are less complicated than constructors.
- If we don't call them explicitly (they are called automatically), and there's only one destructor for each object.
- The name of the destructor is the name of the class, preceded by a tilde (~).

- A destructor, as the name implies, is used to destroy the objects that have been created by a constructor.
- A destructor never takes any arguments nor does it return any value.
- It will be **invoked implicitly by the complier** upon exit from the program(or block or function) to clean up the storage that is no longer accessible.
- It is a **good practice to declare destructors** in a program since it releases memory space for future use.
- Objects are destroyed in the reverse order of creation.
- Destructors receive no parameters and returns no value. Class may have only one destructor (no overloading of destrcutor is allowed)

```
// Destructor (order of invocation) invoked implicitly by the compiler
//Only one default constructor per class
#include <iostream>
using namespace std;
int count = 0;
class CRectangle {
    public:
    CRectangle (int =0, int=1); // prototype of parametrized constructor with default
    ~CRectangle ();//prototype of destructor
    int area (void) {return (width*height); }//member function declared and defined in private:
    int width, height;
};
CRectangle:: CRectangle (int a, int b) //parameterized constructor (no return value)
{
::count++;
cout<<"No."<<::count<<" object created"<< endl;
width= a; height = b;}
CRectangle::~CRectangle () //Destructor
{ cout<<"No."<<::count--<<" object destroyed\n";
}

 int main() {
 cout<<"\n\nENTER MAIN()\n";
 CRectangle rectA (3,3), rectB (2), rectC, rectD;//parameterized constructor invokes
 cout<<"\n\nENTER BLOCK 1\n";
 CRectangle rectE; //default constructor is invoked
 cout<<"\n\nENTER BLOCK 2\n";
 CRectangle rectF; //default constructor is invoked
 cout<<"\n\nRE-ENTER MAIN()\n";
 return 0;}
```

```
ENTER MAIN()
No.1 object created
No.2 object created
No.3 object created
No.4 object created


ENTER BLOCK 1
No.5 object created


ENTER BLOCK 2
No.6 object created


RE-ENTER MAIN()
No.6 object destroyed
No.5 object destroyed
No.4 object destroyed
No.3 object destroyed
No.2 object destroyed
No.1 object destroyed

Process returned 0 (0x0)
Press any key to continue.
```

Concept of Static Variable

- A static variable is a variable in computer programming that is allocated during compilation and remains in existence for the entire program run.
- When a variable is declared as static, space for it gets allocated for the lifetime of the program.
- Even if the function is called multiple times, space for the static variable is allocated only once and the value of the variable in the previous call gets carried through the next function call.
- This is useful for implementing coroutines in C/C++ or any other application where the previous state of function needs to be stored.

Example:

```
//Program without static variable
//Static variable is visible inside the function in which it is defined
//but remains in existence for the life of the program
#include<iostream>
using namespace std;
float getavg (float); // function prototype
int main()
{
float data = 1, avg;
while (data !=0)
{
cout<<"Enter a number: ";
cin>>data;
avg=getavg (data);
cout<<"New average is "<< avg <<endl;}
system ("pause");
 return 0;}
```

```
Enter a number: 20
New average is 20
Enter a number: 50
New average is 50
Enter a number: 30
New average is 30
Enter a number: 20
New average is 20
Enter a number:
```

```
// function definition getavg()
float getavg (float ndata)
{
int Count = 0;
float total = 0; // variables are declared
Count++;
total += ndata;
return (total/Count);
}
```

Example: Use of Static Variable

```
//Program without static variable
//Static variable is visible inside the function in which it is defined
//but remains in existence for the life of the program
#include<iostream>
using namespace std;
float getavg (float); // function prototype
int main()
{
float data = 1, avg;
while (data !=0)
{
cout<<"Enter a number: ";
cin>>data;
avg=getavg (data);
cout<<"New average is "<< avg <<endl;}
system ("pause");
 return 0;}
```

```
Enter a number: 20
New average is 20
Enter a number: 30
New average is 25
Enter a number: 50
New average is 33.3333
Enter a number: 20
New average is 30
Enter a number:
```

```
// function definition getavg()
float getavg (float ndata)
{
static int Count = 0;
static float total = 0;
// static variables are declared only once per program
// count and total increase with each function call
Count++;
total += ndata;
return (total/Count);
}
```

Static Class members

- A static data member is declared by the word **static.**
- Such a member is shared by all the objects of the class – it is common to all objects of that class.
- It exists even when no object of the class has been created.
- A static member needs initialization outside the class.
- It can be private or public
  - private static is accessed through public member functions or friend functions
  - public static is accessed through objects of the class or through the class name using the scope resolution operator

```
// Destructor (order of invocation) invoked implicitly by the compiler
//Only one default constructor per class
//Static variable is implemented in OOP rather than global variable (however it also seems to be global)
 #include <iostream>
using namespace std;
class CRectangle {

public:
CRectangle (int =0, int=1); //prototype of parametrized constructor with default arguments (default constructor)
~CRectangle ();//prototype of destructor

int area (void) {
    return (width*height);
    }//member function declared and defined inside the class private:

static int count; // static variable count as private member (common to all objects)
int width, height; };
 CRectangle:: CRectangle (int a, int b) //parameterized constructor (no return value)
 { count++;
 cout<<"No."<<count<<" object created"<< endl;
 width= a; height = b;}
 CRectangle::~CRectangle () //Destructor
{cout<<"No."<<count--<<" object destroyed\n";}
int CRectangle:: count; // static variable gets initialized to zero by default (otherwise to any given value)
```

```
int main()
{
cout<<"\n\nENTER MAIN()";
CRectangle rectA (3,3), rectB (2), rectC, rectD;//parameterized constructor invoked
cout<<"\n\n ENTER BLOCK 1\n";
CRectangle rectE; //default constructor is invoked
cout<<"\n\nENTER BLOCK 2\n";
CRectangle rectF; //default constructor is invoked
cout<<"\n\nRE-ENTER MAIN ()\n";


cin.get(); cin. ignore ();
return 0;}
```

```
ENTER MAIN()No.1 object created
No.2 object created
No.3 object created
No.4 object created


 ENTER BLOCK 1
No.5 object created


ENTER BLOCK 2
No.6 object created


RE-ENTER MAIN ()


No.6 object destroyed
No.5 object destroyed
No.4 object destroyed
No.3 object destroyed
No.2 object destroyed
No.1 object destroyed

Process returned 0 (0x0)   execution time : 2.628 s
Press any key to continue.
```

Static member function

Static member functions can only access static member variables. Non-static members can not be accessed.

Because non-static member variables must belong to a class object, and static member functions have no class object to work with.

- Static member functions are not attached to an object, they have no **this** pointer.
- The **this pointer** always points to the object that the member function is working on.
- Static member functions do not work on an object, so the this pointer is not needed

Thanks