Constructors and Destructors

- Encapsulations
- Constructors
  - Default Constructors
  - Parameterized Constructors
  - Copy Constructors
  - Dynamic Constructors
- Multiple Constructors in Class
- Constructors with Default Arguments
- Destructors

Encapsulation

- In normal terms Encapsulation is defined as wrapping up of data and information under a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them.

Encapsulation

```
class Encapsulation
{   private:
        int x;
    public:
        void getdata(int a)
        {
            x =a;
        }
        int putdata()
        {
            cout<<x;
        }
};
```

```
int main()
{
    Encapsulation obj;

    obj.getdata(5);

    obj.putdata();
    return 0;
}
```

Constructors

There are mainly four types of COnstructors

- Default Constructors
- Parameterized Constructors
- Copy Constructors
- Dynamic Constructors

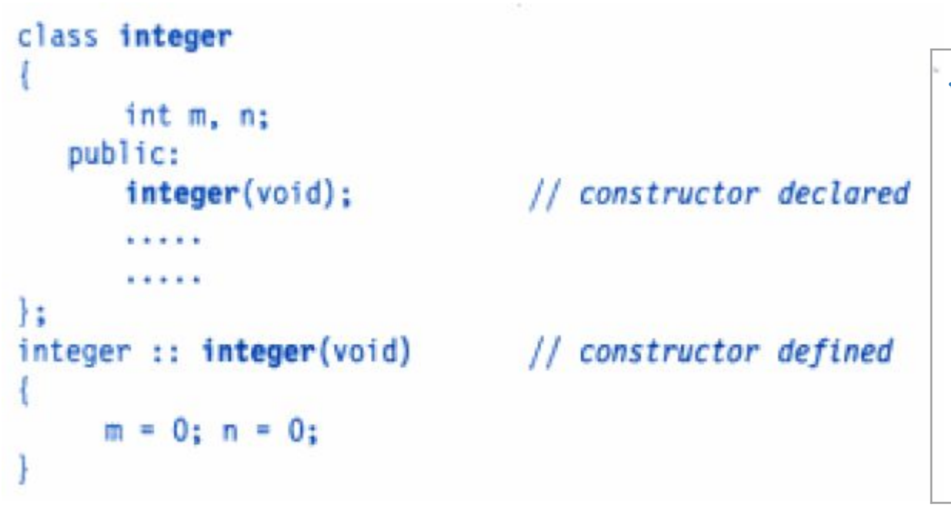
Constructors

The constructor function have some special characteristics

- They should be declared in public section
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefor, they can not return values.
- They can not be inherited, through a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- They make ‘implicit calls’ to the operator ***new*** and ***delete*** when memory allocation is required.
- When a constructor is declared for a class, initialization of the class object becomes mandatory.

Default Constructors

A constructor is a special member function whose task is to initialize objects of its class.It is special because its name is same as class name. A constructor is invoked whenever objects of its class is created



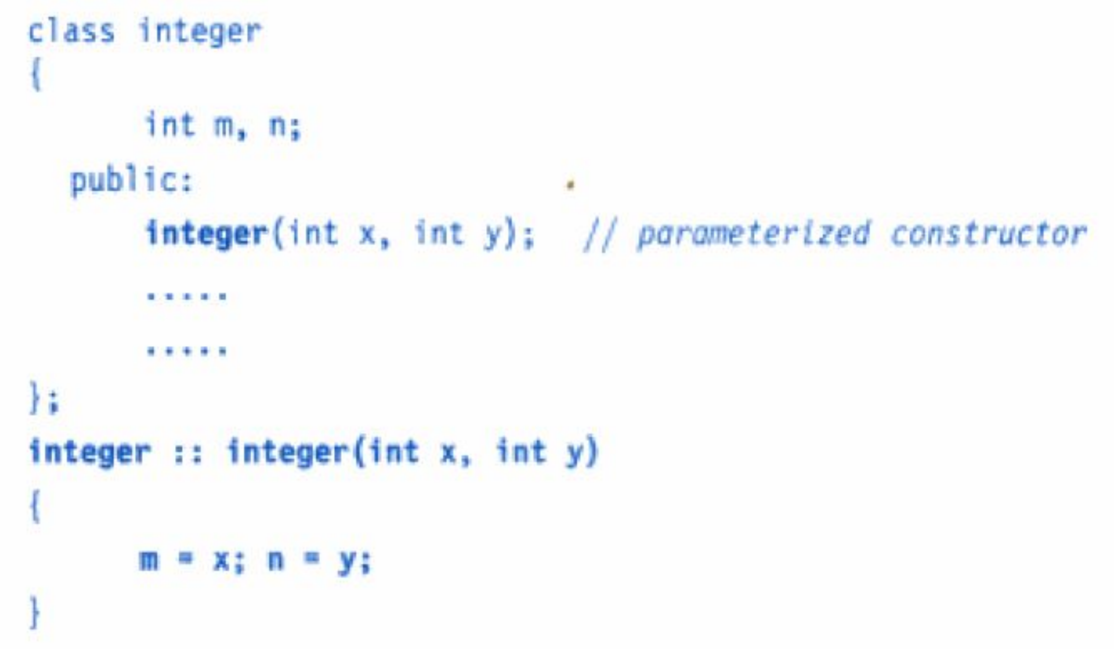
```
class integer
{
        int m, n;
    public:
        integer(void);              // constructor declared
        .....
        .....
};
integer :: integer(void)            // constructor defined
{
      m = 0; n = 0;
}
```

When a class contains a constructor, it is guaranteed that an object created by the class will be initialized automatically. For example,

integer int1

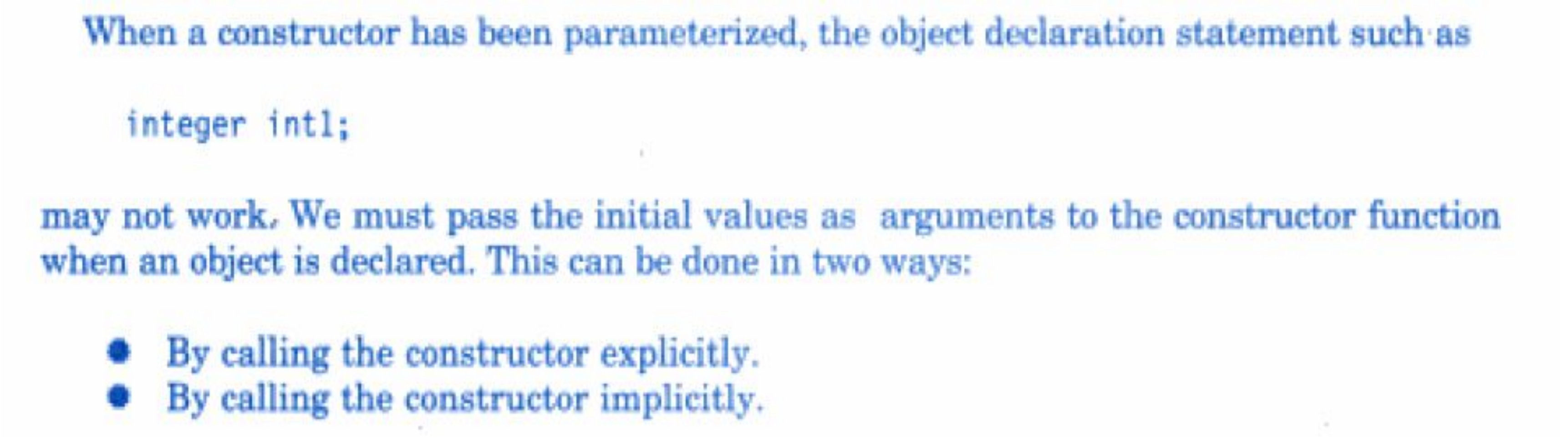
Parameterized Constructors

A 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 Constructors



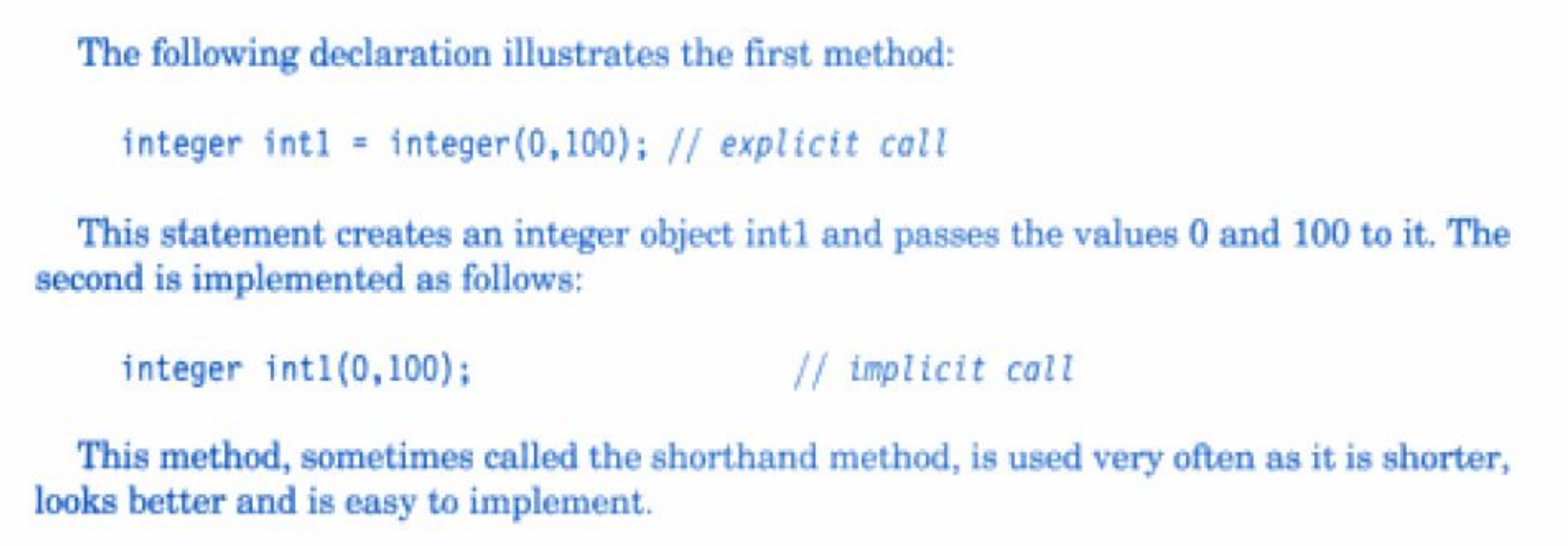
When a constructor has been parameterized, the object declaration statement such as

```
integer int1;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

Parameterized Constructors



The following declaration illustrates the first method:

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

This statement creates an integer object int1 and passes the values 0 and 100 to it. The second is implemented as follows:

```
integer int1(0,100);                    // implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Parameterized Constructors

```
#include<iostream>
using namespace std;
class integer
{
        int m,n;
    public:
        integer(int,int);
        void display()
        {
            cout<<"m = "<<m<<"\n";
            cout<<"n = "<<n<<"\n";
        }
};

integer::integer(int x,int y)
{
    m=x;    n=y;
}
```

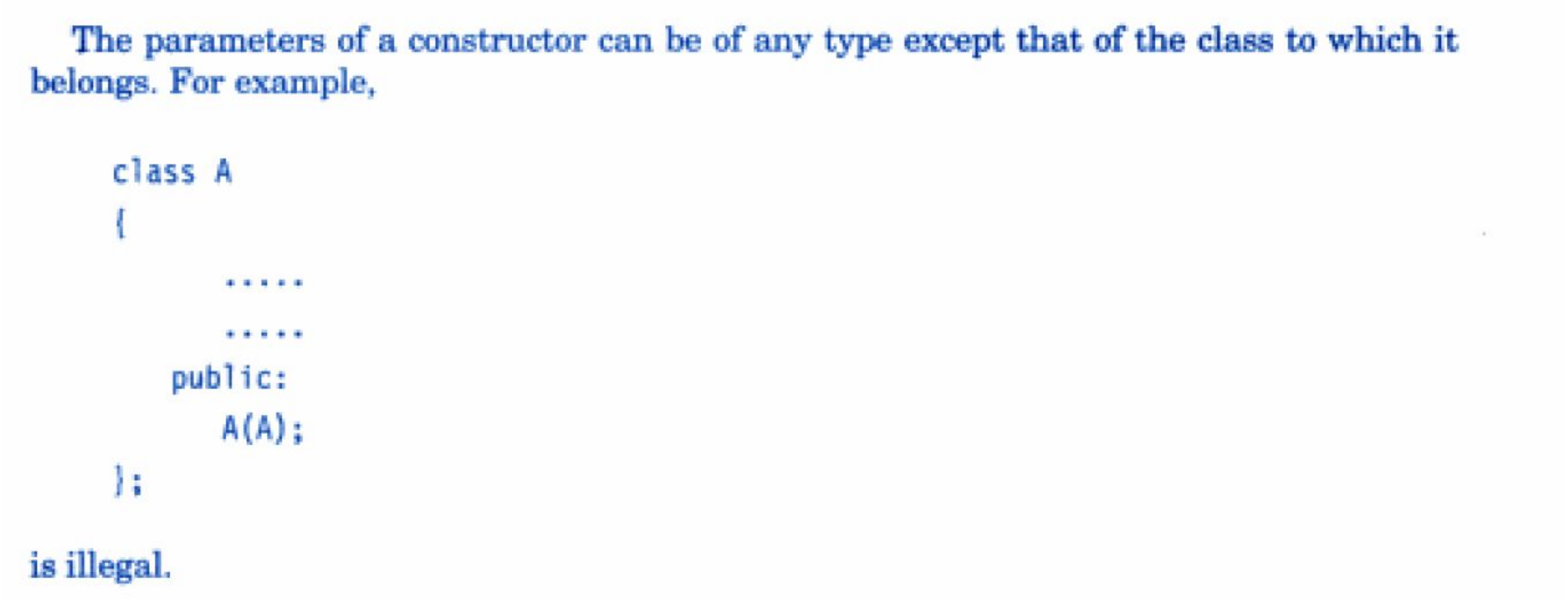
```
int main()
{
   //Constructor call Implicitly
   integer int1(0,100);

   //Constructor call Explicitly
    integer int2=integer(25,75);

    cout<<"\nObject1"<<"\n";
    int1.display();

    cout<<"\nObject2"<<"\n";
    int2.display();
    return 0;
}
```

Copy Constructors

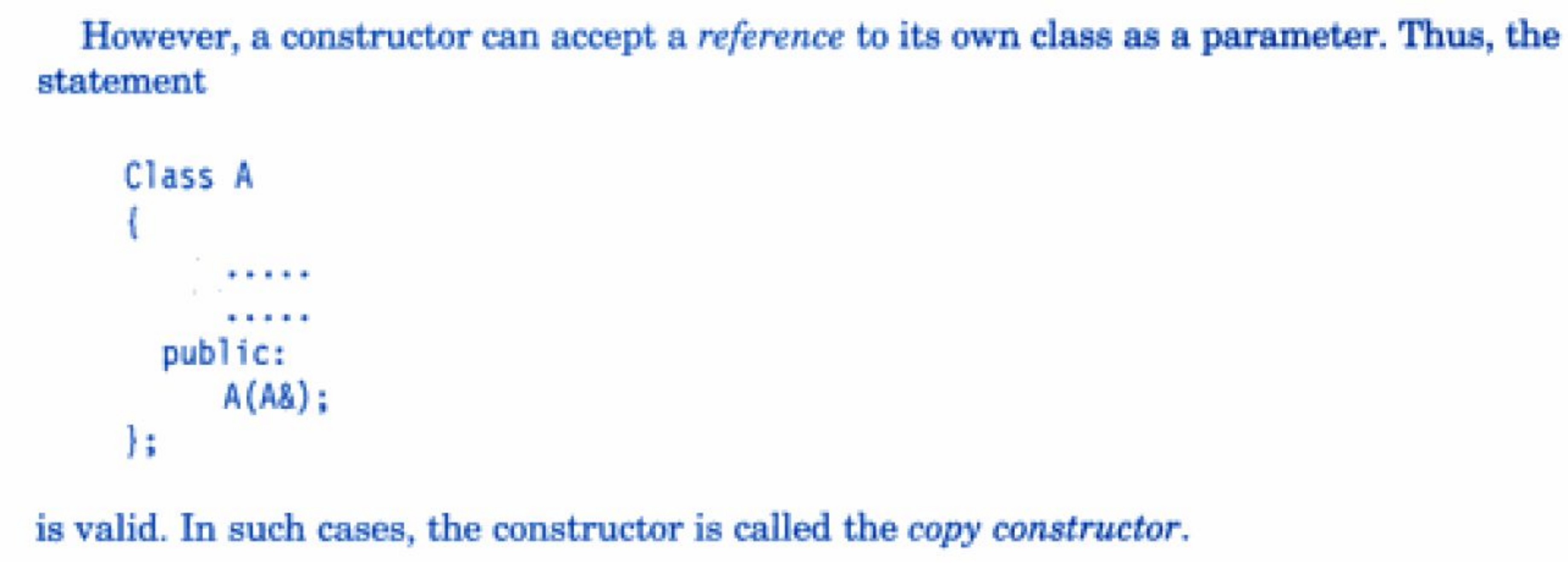


The parameters of a constructor can be of any type except that of the class to which it belongs. For example,

```
class A
{
      .....
      .....
   public:
      A(A);
};
```

is illegal.

Copy Constructors



However, a constructor can accept a *reference* to its own class as a parameter. Thus, the statement

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

is valid. In such cases, the constructor is called the *copy constructor*.

Copy Constructors

Consider the class integer,

```
integer(integer &i);
```

So, the copy constructor is used to declare and initialize an object from another object, for example, the statement

```
Integer I2(I1);
```

Would define the object I2 and at the same time initialize it to the values of I1. Another form of this statement is

```
Integer I2=I1
```

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

Copy Constructors

Remember, the statement

I2=I1

Will not invoke the copy constructor.

However, if I1 and I2 are objects, this statement is legal and simply assign the values of I1 to I2, member-by-member. This is the task of overloaded operator(=).

Copy Constructors

```
class integer
{       int id;
    public:
        integer(int a)
        {   id=a;     }
        integer(code &x)
        {   id=x.id;  }
        void display()
        {   cout<<id<<endl;
        }
};
```

```
int main()
{   integer A(100);
    integer B(A);
    integer C=A;
    integer D;
    D=A;
    A.display();
    B.display();
    C.display();
    D.display();
    return 0;
}
```

Dynamic Constructors

- Dynamic constructor is used to allocate the memory to the objects at the run time.
- Memory is allocated at run time with the help of 'new' operator.
- By using this constructor, we can dynamically initialize the objects.

Dynamic Constructors

```
class sample
{
    char* p;
public:
    // default constructor
    // Also called Dynamic Constructor
    sample()
    {   // allocating memory at run time
        p = new char[20];
        p = "College of Engineering Pune";
    }
```

```
void display()
    {   cout << p << endl;
    }
};

int main()
{   sample obj;
    obj.display();
}
```

Dynamic Constructors

```
class sample  {
    int* p;
public:
    sample()// default constructor
    {   // allocating memory at run time
       // and initializing
        p = new int[3]{ 1, 2, 3 };
        for (int i = 0; i < 3; i++) {
            cout << p[i] << " ";
        }
        cout << endl;
    }
};
```

```
int main()
{


    // five objects will be created
   // for each object
   // default constructor would be called
   // and memory will be allocated
   // to array dynamically
    sample* ptr = new sample[5];
}
```