Object Oriented Programming (CS1004)

Lecture 8

Access Specifier and Constructors

Member Access Specifiers

- **`private`**

- Default access mode
- Data only accessible to member functions and **`friends`**(will be covered later) • **`private`** members only accessible through the **`public`** class interface using **`public`** member functions.
- Declaring data members with access specifier private is known as ***data hiding***. • When a program creates (instantiates) an object, its data members are encapsulated (hidden) in the object and can be accessed only by member functions of the object's class.

- **`public`**
  - Presents clients (or main() function) with a view of the services the class provides
  - Data and member functions are accessible, i.e., Indicates that the data/function is "available to the public"—that is, it can be called by other functions in the program (such as main), and by member functions of other classes (if there are any).

Modified *DayofYear* Class

Example • Data now private

• Objects have no direct access

```
Class DayofYear {
Public:
void input() {
        cout<<"Please enter day: ";
        cin>>day;
        cout<<"Please enter Month: ";
        cin>>month;
 }
        void output() {}
Private:
        int month;
int day;
}
```

• Declare object in main():

DayOfYear today;

• Object today can ONLY access public

members • cin >> today.month; // NOT

ALLOWED! • cout << today.day; // NOT

ALLOWED! • Must instead call public

operations: • today.input();

• today.output();

Member Functions

- Inline
  - The functions are defined within the body of the class definition.

- Out-of-line:
  - The functions are declared within the body of the class definition and defined outside.

Inline member function

- For non-member functions:
  - Use keyword inline in function declaration and function heading
- For class member functions:
  - Place implementation (code) for function IN class definition → automatically

inline

- Use for very short functions only
- Code actually inserted in place of call
  - Eliminates overhead
  - More efficient, but only when short!
  - If too long → actually less efficient!

Member Functions

- If a member function is defined outside the class • Scope resolution operator (::) and class name are needed • Defining a function outside a class does not change it being **public** or **private**
- Binary scope resolution operator (**::**)
  - Combines the class name with the member function name
  - Different classes can have member functions with the same name

- Format for defining member functions

ReturnType ***ClassName::MemberFunctionName****( ){*

…

}

Class exercise

- Create a simple class named ‘Time’. The time class has three attributes declared as private:
  Seconds, minutes, hours,
- in addition it has two methods declared as public,
  display() → displays current time
  input() → ask user to input time
- Define both member functions outside the class
- In main() create two objects of Time, named, noon and

midnight. • Using objects invoke the corresponding methods.

Initializing Objects using constructors

- Constructor is a special *class method* that is automatically called when a class creates its object
  - A constructor has same name class
  - Does not return a value, not even void
  - Initialize some or all member variables
  - Other actions possible as well
- C++ automatically ***calls*** a constructor for each object that is created, which helps ensure that objects are initialized properly before they're used in a

program.

```
     class DayofYear {
          private:
                  int day;
                  int month;
                  int year;
     Public:
          void output() { }
     }
 Int main() {
          DayofYear D;

 }
```

Default Constructor is called.

Lecture’s contents

- Default Constructors
- Parametrized Constructors
- Overloading Constructors with default

arguments • Accessor and mutator functions

• Member initializer syntax

Default Constructors

- Constructor is a function in every class which is called when class creates its object
  - Basically it helps in initializing data members of the class
  - A class may have ***multiple constructors***
- If a class does not *explicitly* include a constructor, the compiler provides a ***default constructor***:
  - Default constructor is a constructor with no arguments, such as: `DayofYear() {}`

- No data members initialized
- Do not rely on compiler-generated constructors

Class definition with default constructor

- Default Constructor is called when an object is declared without any arguments, such as:

```
class DayofYear {
Private:
Int day;
Int month;
Int year;
public:
     DayofYear (){ //constructor explicitly provided by class
          cout << "DayofYear's class object is created";
        } // default constructor
};
void main (void) {
     DayofYear D1;
}
```

Program Output:

```
DayofYear's class object is created
```

Default constructor

- A default constructor is defined as a constructor with no arguments • If a default constructor is provided by the user/compiler then:

```
DayofYear D1; //That's Okay
DayofYear D2(); //Not okay.
```

  - Because compiler sees a function declaration/prototype!

- If a default constructor is not defined then it is not possible to create an object as:

```
DayofYear D;
```

- There should always be a default constructor available in order to declare an object as above.

Constructor overloading

- A class may define more than one *constructor,* just like other functions
  - With different parameter lists
- Recall: a signature consists of:
  - Name of function
  - Parameter list
- Provide constructors for all possible argument-lists

- For our DayofYear class example the following are the default constructors and parameterized constructor:
  DayofYear(); //default constructor
  DayofYear(int d, int m);

Constructor overloading: Date Class in C++

```
class DayofYear
 {
public:
      DayofYear() { cout<<… } //default constructor
      DayofYear (int d, int m ) //overloaded constructor
      {
        day = d;
        month = m;
      }
      void input();
      void output();
   private:
      int day;
      Int month;
};
void main() {
DayofYear today; //uses default constructor
DayofYear birthday(5,3);
}
```

Default Constructor

- If you do not code any constructors for your class, the compiler will generate a "default" constructor without any parameters.
- The compiler-supplied constructor does nothing (effectively it has an empty function

```
class DayofYear
```

{ y).
bod

- If you write any constructors for your class, the compiler will not supply this default

public:

constructor.

```
      //DayofYear() { cout<<… } //default constructor
      DayofYear (int d, int m ) //overloaded constructor
      {
       day = d;
       month = m;
      }
      void input();
      void output();
   private:
      int day;
      Int month;
 };
void main() {
//DayofYear today; //uses default constructor. Error. Because default
 does not exist.
DayofYear birthday(5,3);
```

Constructor with default arguments

- Like other functions constructors can specify *default arguments*.

- The default arguments to the constructor ensure that, even if no values are provided in a constructor call, the constructor still initializes the data members.
- Example of constructors with default arguments are:

  Time::Time(int hour=0, int min=0, int second =0);

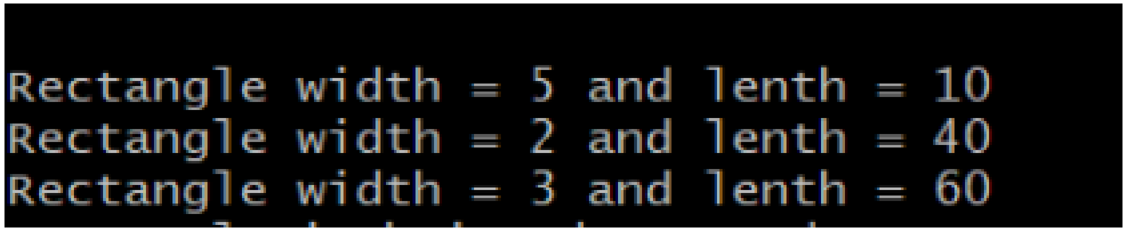
  Date::Date(int day=1,int month=1,int year = 2000);

- A constructor that defaults all its arguments is also a default constructor • That is a constructor that can be invoked with no arguments.

Rectangle class containing constructor with default arguments

```
class Rectangle {
```

```
public:
Rectangle(int w=5,int l=10){//constructor with default
arguments Width=w;
length=l;
  }
private:
 int width;
 int length;
};

int main() {

Rectangle r1;
Rectangle r2(2,40);
Rectangle r3(3,60);

 }
```

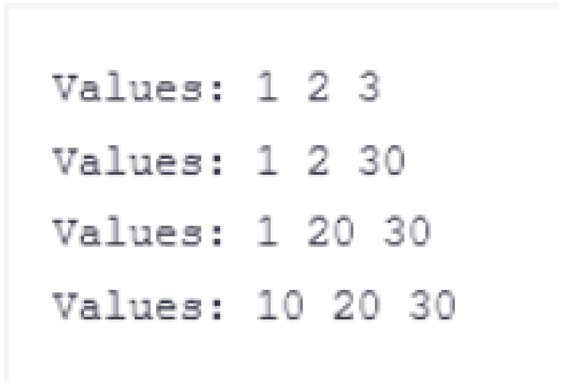


```
Rectangle width = 5 and lenth = 10
Rectangle width = 2 and lenth = 40
Rectangle width = 3 and lenth = 60
```

Example: Multiple Default Arguments from non-oop

- A function can have multiple default arguments:

```
#include <iostream>

void print(int x=10, int y=20, int z=30) {

    std::cout << "Values: " << x << " " << y << " " << z << '\n’;

}

int main() {

print(1, 2, 3); // all explicit arguments

print(1, 2); // rightmost argument defaulted

print(1); // two rightmost arguments defaulted

print(); // all arguments defaulted return 0;

}
```



```
Values: 1 2 3
Values: 1 2 30
Values: 1 20 30
Values: 10 20 30
```

xampe: utpe eaut rguments rom non oop

- Consider function with multiple default arguments from previous slide:

```
void print(int x=10, int y=20, int z=30) {
cout << "Values: " << x << " " << y << " " << z <<
'\n'; }
```

- C++ does not support a function call syntax such as print(,,3) (as a way to provide an explicit value for z while using the default arguments for x and y.
- Some Facts about the use of default arguments:
- (1) Default arguments can only be supplied for the rightmost parameters. The following ***is not allowed***:

```
void print(int x=10, int y); // not allowed
```

- (2) If more than one default argument exists, the leftmost default argument should be the one most likely to be explicitly set by the user. Such as:

```
print(1); //leftmost argument specified.
```

Class exercise (cont'd from the previous)

- Create a simple class named 'Time'. The time class has three attributes declared as private:
  Seconds, minutes, hours,
- in addition it has two methods declared as public,
  display() → displays current time
  input() → ask user to input time.
- Also define a constructor with three default arguments for seconds, minutes and hours, all set to 0
- In main() create two objects of Time, named, noon and

midnight. • Using objects invoke the corresponding methods.

Constructor initializer list

- Initialization lists are an alternative technique for initializing an object's data members in a constructor. For instance, the DayofYear constructor

```
DayofYear::DayofYear(int m, int d, int y)
{
    month = m;
    day = d;
    year = y;
}
```

Can be written using initializing list as:

```
DayofYear::DayofYear(int m, int d, int y) : month(m),
day(d), year(y)
 {
 }
```

Mutator and Accessor (Setter/Getter) Functions

- Every class has member functions that modify the data members, called mutator functions (Setter). For example, in our DayofYear example, we may have a setter function to set the date which is a private member of the class:

```
void setDate(int d);
```

- The setter function does not return anything and usually takes one or more arguments.
- The arguments passed are usually validated inside the setter function.

- Likewise, every class has member functions that only access and do not modify the data members; these methods are called accessor functions (Getter). For example, in our DayofYear example, we may have a getter function to get the year:

  int getYear();

- The getter function does not take an argument and usually returns by value so a client can get the value saved in an object but cannot change the value. Later we will cover how to protect the getter function from accidentally changing the private data members.

Mutator and Accessor (Setter/Getter) Functions

```
class DayofYear {
public:
DayofYear() { } //default constructor
Void setDate(int d){ day=d;}
Int getYear() { return year; }
void input();
void output();
private:
int day;
Int month;
Int year;
};
int main() {
DayofYear today;
today.setDate(28);
Cout<<today.getYear();
}
```

Constructing Arrays of Objects

- We can create arrays of objects in the same fashion as other

  arrays. **`Employee emparray[10];`**

```
DayofYear daysarray[20];
```

- However, declaring the arrays of objects as above will call the default constructor. There must be a default constructor available.

```
class Rectangle {
public:
Rectangle(int w, int l);
//Assume there is no default constructor
private:
 int width;
 int length;
};
int main() {
Rectangle r1[10]; //ERROR: default constructor
Rectangle * r2 = new Rectangle[10]; //ERROR: default constructor }
```

explicit initialization of arrays

- Call to constructors with arguments (non-default) for array members can be made using *explicit initialization of arrays.*

```
class Rectangle {
public:
Rectangle(int w, int l);
private:
 int width;
 int length;
};
int main() {
Rectangle recArray[3] = {
           Rectangle(2,3),
           Rectangle(8,9),
           Rectangle(20,10) };
}
```

Array members calling member functions

- Array members who are objects can call member functions in a usual way:

```
class Rectangle {
public:
Rectangle(int w, int l);
Void setwidth(int);
Void setlength(int);
private:
 int width;
 int length;
 };
 int main() {
Rectangle recArray[3] = {Rectangle(2,3),Rectangle(8,9),Rectangle(20,10)
 };
recArray[0].setwidth(10);
recArray[1].setwidth(20);
recArray[2].setwidth(30);
 }
```