

Lecture 11

copy constructor

Today's Lecture Contents

- Copy constructor

Constructors and Copy Constructors

- All class definitions include a default constructor.
- The ***default constructor*** does not take any arguments.
- It is used to initialize object data fields when no other arguments are specified.
- Besides a default constructor, a class definition also provides a default ***copy constructor***
- A **copy constructor** is a special constructor that is called **whenever a new object is created and initialized with another object's data.**
- For example:

```
Rectangle r1(1,18); //object r1 with length and width
members; Rectangle r2(r1);
```

- The above statement creates object r2 and initializes it with the r1's data. • It uses the default copy constructor provided by the compiler to make a duplicate copy (r2) of the already existing object (r1).

```
class Rectangle {                                Void printDimensions() {
public:                                         Example: Default copy
Rectangle(int w=5,int l=10) {                  constructor
width=w; length=l;}
```

```
Cout<<"Rectangle Width= "<<width<<" length= "<<length<<endl;
} private:
    int width;
    int length;
};
int main() {
```

```
Rectangle r1(10,40);  
Rectangle r2(r1); //creates and initializes r2 by calling default  
copy constructor  
r1.printDimensions();  
r2.printDimensions();  
}
```

```
Rectangle width = 10 and lenth = 40  
Rectangle width = 10 and lenth = 40
```

Non-default Copy Constructor

- Like a normal constructor it is also possible to explicitly provide a copy constructor. • In the presence of the user-provided copy constructor, the default copy constructor will not be called.
- Most of the time, the default copy constructor is sufficient, but there will be situations where we will need an explicit copy constructor.
- Signature of the copy constructor may look something like this:
`Rectangle::Rectangle(const Rectangle &obj);`
- And its full implementation may look something like this:

```
Rectangle::Rectangle(const Rectangle &obj) {  
    width = obj.width;  
    length = obj.length;  
}
```

```
class Rectangle {  
public:  
    Rectangle(int w=5,int l=10) {  
        width=w; length=l;}  
};
```

```
Rectangle(const Rectangle &obj) {
```

Example: non-Default copy constructor

```
    width = obj.width; length = obj.length;
```

```
cout<<"Rectangle created using non-default copy constructor: ";
```

```
Void printDimensions() {
```

```
    Cout<<"Rectangle Width= "<<width<<" length= "<<length<<endl; }
```

```
private:
```

```
    int width;
```

```
    int length;
```

```
};
```

```
int main() {
```

```
Rectangle r1(10,40);  
Rectangle r2(r1); //creates and initializes r2 by calling copy  
constructor  
r1.printDimensions();  
r2.printDimensions();  
}
```

```
Rectangle created using non-default copy constructor:  
    Rectangle width = 10 and lenth = 40  
  
    Rectangle width = 10 and lenth = 40
```

when is copy constructor called?

- When is a copy of an object made?
 - Passing object by value as a parameter.

```
void displayRectangle(Rectangle R);
```

- returning an object from a function by value.

```
Rectangle createRectangle();
```

- Constructing one object based on another of the same class.

```
Rectangle r1;
```

```
Rectangle r2(r1);
```

Shallow copy

- The default copy constructor simply copies the bytes of the object.
 - This is called a **shallow copy**.
- As long as all of your object's data is inside the object, a **shallow copy** is fine.
 - For example, we used default copy constructor to make copy of our **Rectangle class object**.
- However, if an *object has a pointer to dynamic storage* as a data member, not all of its data is inside the object.
- The dynamic storage is separate from the object itself, and resides on the free store.
- A **shallow copy** will simply copy the pointer to this dynamic storage without actually making a copy of the storage contents.

- The result is that you end up with two different objects pointing to the same chunk of dynamic storage.

Example: Shallow copy

```
class dynamicvar {
private:
    int * ptr;
public:
    dynamicvar(int n) {
        cout<<"in constructor: Allocating variable on the heap: "<<endl;
        ptr = new int;
        *ptr = n;
    }
    ~dynamicvar() {
        cout<<"In Destructor: Dynamically allocated variable is being deleted:";
        delete ptr; }
};

int main() {
    dynamicvar D1(5);
    D1.display();
    {
```

```

        Dynamicvar D2(D1); //calling default copy constructor
        D2.modify(10);
        D1.display(); //what will it print?
    } //destructor called for D2.
D1.display()
}

```

User defined copy constructor

- The below user-defined copy constructor does *deep copy*, instead of a shallow copy.

```

class dynamicvar {
public:
    dynamicvar(int n) {
        cout<<"in constructor: Allocating variable on the heap: "<<endl;
        ptr = new int;
        *ptr = n;
    }

    dynamicvar(const dynamicvar & obj) {
        cout<<"in copy constructor: copying object: "<<endl; ptr =

```

```

new int;
*ptr = *(obj.ptr);
}

~dynamicvar() {
cout<<"In Destructor: Dynamically allocated variable is being deleted:";
delete ptr; }
};

```

Shallow Copy: Another Example

- Default copy constructors work fine unless the class contains pointer data members.
- For example, If name had a dynamically allocated array of characters (i.e., one of the data members is a pointer to a char).

```

#include <string.h>
class name {
public:
    name(const char* = ""); //default constructor    ~name()
    { delete ptr; } //destructor

private:
    char* ptr; //pointer to name

```

```

    int length; //length of name including null char
};

name::name(const char* name_ptr) { //constructor
length = strlen(name_ptr); //get name length
    ptr = new char[length+1]; //dynamically allocate
strcpy(ptr, name_ptr); //copy name into new space }

Int main() {
Student smith("Sue Smith"); //one argument
constructor Student clone(smith); //using default copy
constructor

}

```

Copy Constructors

- the following shallow copy is disastrous!

```

n a m e s m i t h ("S u e S m i t h "); // o n e a r g c o n s t r u c t o r u s e

```

```

d n a m e c l o n e ( s m i t h ); // d e f a u l t c o p y c o n s t r u c t o r u s e
d

```

```
graph LR; smith --> ptr; ptr --> clone; ptr --> S; length=10 --> h; u --> m; e --> i; t
```

Defining your own Copy Constructors

- To resolve the pass by value and the initialization issues, we must write a copy constructor whenever dynamic member is allocated on an object-by-object basis. → This copy constructor makes a "deep copy" of the object
- A copy constructor takes an instance of the same class as a constant reference argument.

- They have the form:

```
class_name(const class_name &class_object);
```

- Notice the name of the “function” is the same name as the class, and has no return type
- The argument’s data type is that of the class, passed as a constant reference

Defining your own Copy Constructors

```
class name {  
    public:  
        name(const char* = ""); //default constructor  
name(const name &); //copy constructor    ~name() { delete  
ptr}; //destructor    private:  
    char* ptr; //pointer to name  
    int length; //length of name including nul char };
```

```
name::name(const char* name_ptr) { //constructor  length
= strlen(name_ptr); //get name length  ptr = new
char[length+1]; //dynamically allocate  strcpy(ptr,
name_ptr); //copy name into new space }
```

```
name::name(const name &obj) { //copy constructor
length = obj.length; //get length
ptr = new char[length+1]; //dynamically allocate
strcpy(ptr, obj.ptr); //copy name into new space }
```

Defining your own Copy Constructors

- Now, when we use the following constructors for initialization, the two objects no longer share memory but have their own allocated

```
Student smith("Sue Smith"); //one argument
constructor Student clone(smith); //
```



length=10	m	e	t
u	i	h	
e	t	S	'\0'
	h	m	length=10
S	'\0'	i	
	u		

How does a member function know the object that called it?

- Each C++ object gets its own copy of each of the **non-static data members** of the class.
 - These data members are sometimes referred to as instance variables since they are a property of each instance / object.)
- All objects of a class share a single copy of each **static data member** of the class.
 - These data members are sometimes referred to as class variables since they are a property of the class as a whole rather than a particular object of the class.

- A single copy of each compiled **member function definition** is stored in the code segment of the C++ program. – how does the code in that member function access the non-static data members for the correct object?

this pointer

- Suppose there is a class named Student
- `Student st1, st2, st3;`
- Member function space is common for every object of a class
- Whenever a new object is created:
 - Memory is reserved for data members only
 - Previously defined member functions are used over and over again

rollNo, ...

st2

rollNo, ...

st1

Space
getRollNo(),
...
st3
rollNo, ...
Function

this pointer

- When a member function is called, how does it know on which object to act on?
 - The compiler supplies a special implicit pointer named `this` that allows a member function to access the non-static data members.
 - The **this** pointer contains the address of the object that called the member function.
 - The **this** pointer is passed as an extra, hidden argument whenever a non-static member function is called and is available as a local variable within the body of all non-static member functions.

- The **this** pointer is not available in static member functions since they can be called using a class name rather than the name of a specific object.

How **this** pointer is passed to the function?

- The this pointer is passed as an extra, hidden argument whenever a non static member functions is called and is available as a local variable within the body of all non-static member functions.
- 'This' pointer contains the 'self-address' of the object. This address is dereferenced by the functions and hence they act on correct objects • For example,

```
student s1("Ali");
student s2("Ahmed");
```

```
s1.setID(1); //Passes &s1 as the this pointer for setID().
s2.setID(2); //Passes &s2 as the this pointer for setID().
```

	st1	st2	rollNo, ...
	rollNo, ...	rollNo, ...	rollNo, ...
		st3	

address address address

Function
Space
getRollNo(), ...

Using the this Pointer inside member function

- Most uses of the this pointer inside a member function's body will be implicit. Such as:

```
void student::setID(int i) {  
    ID = i;  
}
```

- We can also explicitly use `this` pointer:

```
void student::setID(int i) {  
    this->ID = i;  
    //OR  
    (*this).ID = i;  
}
```

implicitly and explicitly using this pointer

```
class Student {  
    private:  
    string name;  
    int rollNo;  
    int age;  
    public:  
    Student(string aName="", int aNo=1,int  
myage=20):name(aName),rollNo(aNo),age(myage) {} void  
print() {  
    //implicitly use this pointer
```

```
cout<<"Student age is: "<<age<<endl;
//explicitly use dereferenced this pointer
cout<<(*this).age<<endl;
//explicitly use this pointer with arrow operator.
cout<<this->age<<endl;
}
};
int main() {
    Student s1;
    s1.print();
    return 0;
}
```

Using the this Pointer inside member function

- If you have a local variable in a member function that has the same name as a data member of the class, using the `this` pointer explicitly will allow the compiler to tell them apart: Such as:

```
void student::setID(int ID) {  
    this->ID = ID;  
}
```

-

Returning the this Pointer for cascaded function calls

- There are situations where the programmer wants to return reference to current object from a function
- In such cases reference is taken from this pointer like (*this)
 - Pointer returned from the member function can be used for the cascaded function calls

```
{  
    //..body of the member function  
    return *this;  
}
```

- Cascaded member function calls
 - Other functions can operate on that pointer: For example,

S.setName().setAge().setRollNo();

Passing **this** pointer (Example)

```
Student& Student::setRollNo(int aNo)
{
    ...
    return *this;
}
Student& Student::setName(char *aName)
{
    ...
    return *this;
}
int main()
{
    Student aStudent;
    aStudent.setName("John").setRollNo(1024);
}
```

```
...  
    return 0;  
}
```

Cascaded function calls using this pointer

```
class Student {  
private:  
    string name;  
    int rollNo;  
    int age;  
public:  
    Student(string aName="", int aNo=1,int myage=20):  
        name(aName),rollNo(aNo),age(myage) {}  
    Student &setRollNo(int aRollNo) {  
        rollNo = aRollNo;  
        return *this; //enables cascading; }  
    Student &setage(int aAge) {  
        age = aAge;  
        return *this; //enables cascading; }  
    Student &setName(string aName) { name = aName; return
```

```

        *this; //enables cascading; }

};

int main() {
    Student aStudent;
    aStudent.setName("Muhammad").setRollNo(1).setage(30);
    return 0;
}

```

using this pointer to enable cascading function calls

- If a function does not return this pointer, it should be called in the end of the list of the calls. For example,

```
S.setage().printage();
```

Function setage() returns this pointer, but printage() does not return pointer.

```

Student & setage(int a) { age = a; return *this;}
Void printage() { cout<<"student age: "<<age; }

```

Class exercise

- Create a time class, containing, hour, minutes and seconds as data members.
- It also contains, sethour(), setminute() and setseconds() as member functions.
- Now from the main() function using the object of the Time class call the cascaded function calls to set hour, minutes and seconds.