EECS402 Lecture 07

The University Of Michigan

Andrew M. Morgan

Savitch Ch. 7
Objects
Constructors
Destructors

Objects As Members

- Class attributes (i.e. data members) can be of any data type
  - Including user-defined class types

```
class InnerClass
{
  public:
    int inInt;

    void print() const;
};

class OuterClass
{
  public:
    int outInt;
    InnerClass inObj;
    InnerClass inObj2;

    void print() const;
};
```

The outer class has 2 member objects of type InnerClass (in addition to the integer data member "outInt").

Both classes have a function with the same prototype "void print();".

How would you implement both functions outside of the class definitions without ambiguity?

Example Implementation

- While the function prototypes are the same, they are unique in what class they belong to
  - The use of scope resolution can solve the ambiguity problem
  - Remember – scope resolution is often read "belongs to"

This version of print() "belongs to" the class InnerClass

```
void InnerClass::print() const
{
  cout << "InnerClass Int: " << inInt << endl;
}
```

This version of print() "belongs to" the class OuterClass

```
void OuterClass::print() const
{
  cout << "OuterClass Int: " << outInt << endl;
  cout << "Obj: ";
  inObj.print();
  cout << "Obj2: ";
  inObj2.print();
}
```


Using InnerClass and OuterClass

- The dot operator can be used on any object
  - Sometimes, this results in many dots being used in one statement

```
int main()
{
  OuterClass outObj;
  InnerClass inObj;

  outObj.outInt = 15;
  outObj.inObj.inInt = 20;
  outObj.inObj2.inInt = 30;
  outObj.print();

  cout << endl;

  inObj.inInt = 50;
  inObj.print();

  return 0;
}
```

inInt is a data member of inObj, which is a data member of outObj, and is accessed as shown

```
OuterClass Int: 15
Obj: InnerClass Int: 20
Obj2: InnerClass Int: 30

InnerClass Int: 50
```

Constructors & Destructors

- A constructor is a special member function of a class
  - The name of the function is the same as the name of the class
  - It has no return type (can not return a value)
  - It is ***automatically*** called every time a new object is constructed
  - The programmer can not directly "call" the constructor
  - Often abbreviated "ctor"
- A destructor is a special member function of a class
  - The name of the function is the same as the name of the class with a "~" before it
  - It has no return type (can not return a value)
  - It is ***automatically*** called when an object is destroyed
    - When a static object goes out of scope, or a dynamic object is deleted (will discuss later
  - The programmer can not directly "call" the destructor
  - Often abbreviated "dtor"

Ctor And Dtor Purpose

- Why are ctors important?
  - Since they are called automatically every time an object is created, they are usually used to initialize data members for a new object
  - Ctors, like most functions can be overloaded
    - Developer might provide several different ctors (with unique signatures) to initialize members in different ways
  - The "default ctor" is the ctor that takes in no parameters
- Why are dtors important?
  - At the moment, they aren't
  - They are very important when using dynamic allocation, and more detail will be given at that time
  - Dtors, unlike most functions, can ***not*** be overloaded
  - Only one dtor can exist per class!

Ctors and Dtors - Prototypes

- Programmers *do not call* constructors or destructors
- They are called automatically!

```
class IntClass
{
  private:
    int val;

  public:
    IntClass(); //Default constructor
    IntClass(const int inVal); //Ctor to assign val to
                               //a specific value
    ~IntClass(); //Destructor
    void add(const int addVal); //Adds "addVal" to the data member
    void sub(const int subVal); //Subtracts "subVal" from data member
};
```


Ctors and Dtors, Implementation

```
//Default constructor - NO return type (not even void)
//Called when space for an IntClass object is claimed
IntClass::IntClass()
{
  val = 0;
  cout << "IntClass obj constructed!" << endl;
}

//Ctor to assign data member to specified value
IntClass::IntClass(const int inVal)
{
  val = inVal;
  cout << "IntClass obj constructed!" << endl;
}

//Destructor - called when an IntClass object is destroyed
IntClass::~IntClass()
{
  cout << "IntClass obj destroyed - val was: "
       << val << endl;
}
```

Use of Ctors and Dtor


```
int main(void)
{
  IntClass i1;    //Initializes i1.val to 0
  IntClass i2(5); //Initializes i2.val to 5

  cout << "Done with declarations." << endl;
  i1.add(50); //Adds 50 to i1.val, resulting in 50
  i2.add(75); //Adds 75 to i2.val, resulting in 80
  cout << "Reached end of program" << endl;

  return 0;
}
```

```
IntClass obj constructed!
IntClass obj constructed!
Done with declarations.
Reached end of program
IntClass obj destroyed - val was: 80
IntClass obj destroyed - val was: 50
```

Explanation of output lines

i1 gets constructed
i2 gets constructed

i2 gets destroyed
i1 gets destroyed

Fun With Ctors and Dtors

- Recall the InnerClass and OuterClass example from earlier
  - Add ctors and dtors to each with print statements

```
class InnerClass
{
  public:
    int inInt;

    InnerClass()
    {
      cout << "InnerClass obj created!" << endl;
    }

    ~InnerClass()
    {
      cout << "InnerClass obj dest. val: "
           << inInt << endl;
    }
};
```

Fun With Ctors and Dtors, Cot'd

EECS 402

```
class OuterClass
{
  public:
    int outInt;
    InnerClass inObj;
    InnerClass inObj2;

    OuterClass()
    {
      cout << "OuterClass obj created!" << endl;
    }

    ~OuterClass()
    {
      cout << "OuterClass obj dest. val:"
           << outInt << endl;
    }
};
```


Fun with Ctors and Dtors, cot'd

EECS 402

```
int main(void)
{
  InnerClass ic;
  OuterClass oc;

  cout << "Done with decls." << endl;
  ic.inInt = 1;
  oc.outInt = 2;
  oc.inObj.inInt = 3;
  oc.inObj2.inInt = 4;
  cout << "Done with program" << endl;

  return 0;
}
```

```
InnerClass obj created!
InnerClass obj created!
InnerClass obj created!
OuterClass obj created!
Done with decls.
Done with program
OuterClass obj dest. val:2
InnerClass obj dest. val: 4
InnerClass obj dest. val: 3
InnerClass obj dest. val: 1
```

OC

IC

Step through the code to determine what order objects get created/destroyed in this example.

Odd Output???

EECS 402

```
class MyClass
{
  public:
    int val;

    MyClass()
    {
      val = 0;
      cout << "MyClass ctor" << endl;
    }
    ~MyClass()
    {
      cout << "MyClass dtor" << endl;
    }
};

void printMyClass(const MyClass mc)
{
  cout << "MyClass val: "
       << mc.val << endl;
}
```

From main:

```
MyClass myObj;

printMyClass(myObj);
myObj.val = 15;
printMyClass(myObj);
```

```
MyClass ctor
MyClass val: 0
MyClass dtor
MyClass val: 15
MyClass dtor
MyClass dtor
```

Only 1 ctor message printed, but 3 dtor messages printed!

What is happening?

Copy Constructor

EECS 402

- There is another special constructor, often called by default - the copy constructor.
- Its signature looks like this:

```
ClassName(const ClassName &copy);
```

- The copy constructor is called when a new object is needed that is a copy of another object
- There is a "free copy ctor" that is ***automatically generated*** for each class you create
  - This means the function exists, even if the programmer didn't write it!
- The free copy constructor is automatically called when needed
  - i.e. when an object is passed-by-value into a function
  - The free copy ctor simply does a data member-by-member copy from the original object into the new object

Copy Ctor, Cot'd

EECS 402

- You can "override" the free copy ctor by implementing your own
- When you write your own copy ctor, it is called ***in place of*** the free copy ctor
  - Therefore, if you want/need to add functionality to a copy ctor, you need to include the original functionality with the new
- The copy ctor is just another overloaded ctor
  - Anytime any object is created one ctor will get called
  - For value parameters, the new object is created and initialized using this version of the ctor

Copy Ctor, Example

EECS 402

```
class MyClass
{
  public:
    int val;
    MyClass()
    {
      val = 0;
      cout << "MyClass ctor" << endl;
    }
    ~MyClass()
    {
      cout << "MyClass dtor" << endl;
    }
    //This is the overriding copy ctor
    MyClass(const MyClass &copy)
    {
      val = copy.val; //<--default behavior
      cout << "MyClass copy ctor" << endl;
    }
};

void printMyClass(const MyClass mc)
{
  cout << "MyClass val: " << mc.val << endl;
}
```

From main:

```
MyClass myObj;

printMyClass(myObj);
myObj.val = 15;
printMyClass(myObj);
```

```
MyClass ctor
MyClass copy ctor
MyClass val: 0
MyClass dtor
MyClass copy ctor
MyClass val: 15
MyClass dtor
MyClass dtor
```

Constructor Initializers

- Initializing values in constructors is so common, a special syntax was developed
- Constructor initializers only work with constructors!

```
class InitClass
{
  public:
    InitClass():intVar(0), floatVar(0.0)
    {      //initializes intVar to 0 and
    }      //floatVar to 0.0

    InitClass(const int inI, const float inFf):intVar(inI), floatVar(inF)
    {      //initializes members to parameters
    }
    //... More functions..

  private:
    int intVar;
    float floatVar;
};
```


Misc Ctor Information

- When you implement a ctor, the default ctor is no longer available
- Ctors can't be called like other member functions
  - They are only called automatically

```
class XClass
{
  public:
    int a;
};

int main()
{
  XClass xObj;
  xObj.a = 50;
  cout << xObj.a;
  cout << endl;
  return 0;
}
```

Compiles

```
class XClass
{
  public:
    int a;
    XClass(int b):a(b)
    { ; }
};

int main()
{
  XClass xObj;
  xObj.a = 50;
  cout << xObj.a;
  cout << endl;
  return 0;
}
```

Does Not Compile

```
class XClass
{
  public:
    int a;
    XClass(int b):a(b)
    { ; }
    XClass():a(0)
    { ; }
};

int main()
{
  XClass xObj;
  xObj.a = 50;
  cout << xObj.a;
  cout << endl;
  return 0;
}
```

Compiles

```
class XClass
{
  public:
    int a;
    XClass(int b):a(b)
    { ; }
    XClass():a(0)
    { ; }
};

int main()
{
  XClass xObj;
  xObj.XClass(5);
  xObj.a = 50;
  cout << xObj.a;
  cout << endl;
  return 0;
}
```

Does Not Compile

Why Won't This Compile?


```
class InClass
{
  private:
    int i;
  public:
    InClass(const int inI):i(inI)
    { ; }

    void print() const
    {
      cout << "i: " << i;
      cout << endl;
    }
};

//Continued next column
```

```
class OutClass
{
  private:
    int o;
    InClass iObj;
  public:
    OutClass(const int inO):o(inO)
    { ; }

    void print() const
    {
      cout << "o: " << o;
      cout << " iObj: ";
      iObj.print();
    }
};

int main()
{
  OutClass outClassObj(50);
  outClassObj.print();

  return 0;
}
```


Explanation Of Example Program

- The iObj member of OutClass is an object
- In order to create an object, a ctor must be used
- The default ctor is used, since the use of another is not specified
  - The InClass has a ctor defined, though, therefore the default ctor is no longer available
- How about including a default ctor for InClass to fix it?
  - This would work – however, what if you don't want a default ctor available to the user?
  - Perhaps you don't want to let the user to create an InClass object unless they provide a value to initialize the member to!
- Constructor initializers can be used to specify a ctor for initializing members

Updated Program For Compile

```
class InClass
{
  private:
    int i;
  public:
    InClass(const int inI):i(inI)
    { ; }
    void print() const
    {
      cout << "i: " << i;
      cout << endl;
    }
};

//Continued next column
```

```
class OutClass
{
  private:
    int o;
    InClass iObj;
  public:
    OutClass(const int inO):o(inO),iObj(0)
    { ; }
    void print() const
    {
      cout << "o: " << o;
      cout << " iObj: ";
      iObj.print();
    }
};

int main()
{
  OutClass outClassObj(50);
  outClassObj.print();
  return 0;
}
```

This ctor initializer doesn't assign iObj to 0 (check the types – InClass is not an int). Instead, it specifies which version of the ctor is used to create the iObj object.

```
o: 50 iObj: i: 0
```