Constructors--an Introduction

There's an inefficiency with the Course class we looked at last time. In order to "set up" the class with initial data I have to call the "setter" functions manually, like this:

```
void main()
{
    Course cs213;
    cs213.setInstructor("DiNapoli");
    cs213.setStudentCount(45);
    cs213.setCourseName("COM S 213");
    // rest of program here
}
```

Constructors--an Introduction

I could set up an "init" member function that takes three arguments.

```
class Course
{
    public: // These can be seen outside the class
        // init function
    void init(string argName, string argInstructor, int size);
    // Define member functions
    string getCourseName();
    string getInstructor();
    int getStudentCount();
    void setCourseName(string theName);
};
```

Constructors--an Introduction

And define it like this:

```
void Course:: init(string argName, string argInstructor, int size)
{
    name = argName;
    instructor = argInstructor;
    numStudents = size;
}
```

Constructors--an Introduction

Then, whenever I needed to initialize a new instance of a "Course", I could just use the "init" function:

```
void main()
{
    Course cs213;
    cs213.init("COM S 213","DiNapoli",45);
    // rest of program here
}
```

Constructors--an Introduction

In this "init" member function we can do things like:
- zero out member variables (provide initial values)
- allocate dynamic space
- C++ has a built in mechanism to doing this type of work.
- It is called a constructor.
- A constructor is a special member function which is always called immediately after space is allocated for the class instance in question.
- The member function name of the constructor is required to be the same name as the class.
- So, if we had a class named Calculator, we would define the constructor as follows:
Constructors

class Calculator
{
public:
  Calculator();  // Declare the constructor
  bool calculate(char op, float arg1, float arg2, float &result);
  int getOperationsCount() { return opCount; }
private:
  int opCount;
};

// Here's the constructor definition
Calculator::Calculator()
{
opCount = 0;
}

Simple Constructors

Calculator::Calculator()
{
opCount = 0;
}

- Notice a couple of things:
  - The constructor is defined the same way as any other member function
  - Except, there is no return type
  - Inside the constructor we can perform necessary initializations.

- When does a Constructor get called?
  - A constructor gets called when the object is created.
  - Whether the object is created statically (local variable)
  - or dynamically (with the new operator)
  - You do not need to explicitly call the constructor yourself.
- Let's see an example...

Demonstration #1

A Simple Constructor

Constructors with Arguments

- You may define constructors which take arguments as well.
- Consider a simple Course class
  - similar to the one we used earlier

class Course
{
public:
  Course(string theCourseName, string theInstructor,
          int classSize);
private:
  string courseName;
  string instructor;
  int size;
};

- Notice how there is no "init" member function...

Constructors with Arguments

- We would define the Constructor as follows:

  Course::Course(string theCourseName, string theInstructor,
                  int classSize)
  {
    courseName = theCourseName;
    instructor = theInstructor;
    size = classSize;
  }

- This saves us having to define a separate "init" member function
- More importantly, this will be called automatically!
- But if a constructor takes arguments, how do we pass them?
Constructors with Arguments
- There are two ways to call a constructor with arguments:
  - We’ll cover the second way when we go cover pointers

```cpp
int main()
{
    Course cs213("COM S 213","Ron DiNapoli",45);
    // Rest of program here
}
```

- Again, this saves us having to write a separate “init” function
- But can you have a simple constructor declared as well?
- What happens if you do the following...

```cpp
class Course
{
public:
    Course(); // Simple Constructor
    Course(string theCourseName,string theInstructor,
        int classSize); // Constructor with arguments
private:
    string courseName;
    string instructor;
    int size;
};
```

- Can you really have two member functions with the same name but different arguments?
- Yes, you can. It is called Overloading.
- The linker will make sure the right version gets called.

Overloaded Constructors

```cpp
-course::Course()
{
    courseName = "";
    instructor = "";
    size = 0;
}
-course::Course(string theCourseName,string theInstructor,
        int classSize)
{
    courseName = theCourseName;
    instructor = theInstructor;
    size = classSize;
}
```

- If a Course object is created with no arguments specified, the simple constructor is called...

Demonstration #2

Overloaded Constructors

A Simple Number Class
- For today’s lecture, we’ll play with the following Number class

```cpp
class Number
{
public:
    Number();
    Number(int initValue);
    void setBase(int);
    int getBase();
    string printValue();
    void setValue(int);
    int getValue();
private:
    long theValue;
    int base;
};
```
Inline Functions
- Any function declaration may have the optional inline keyword
- A function designated as inline function will have the following behavior:
  - Whenever this function is called the compiler has the option of replacing the call with the body of the actual function.
  - This is, in theory, a way for programmers to optimize code themselves.
  - The compiler may not listen to you:
    - Recursive functions
    - Very complex functions

This is how you designate a function as being an "inline" function:

```c
inline int performAddition(int x, int y)
{
  return x+y;
}
```

Operator Overloading
- In addition to overloading functions, you can also overload operators.
- The following operators may be overloaded:

Unary Operators:
- ++ -- - !
- new new[] delete delete[]

Binary Operators:
- -> * / + - << >> < <= > >= != & ^ | && ||
- = += -= %= += -= <<= >>= &= |= ^=

Assignment Operators:
- = *= /= %= += -= <<= >>= &= |= ^=

You cannot alter precedence, only extend the definition as they apply to the particular class you are overloading them from.

Unary Operator Overloading (cont)
- Just for fun, let's overload the unary ~ to mean string representation of Number, and * to mean integer value.
- To overload, we use the following definition:

```c
string Number::operator~()
{
  return getValueStr();
}
```

```c
int Number::operator*()
{
  return getValue();
}
```

- Let's check it out...

Demonstration #4
Unary Operator Overloads

Binary Operator Overloading
- I can see it now, you're all thinking "COOL, what else can we overload".
- OK, ok, you don't have to twist my arm. How about overloading the binary + to do addition?

```c
inline Number operator+(Number &num1, Number &num2)
{
  // This is somewhat cheating. Let's retrieve the // integer values, add them, then stuff them back // into a "Number" which we return
  Number temp(num1.getValue() + num2.getValue());
  return temp;
}
```

- But why inline? Any why is this defined globally?
Demonstration #5

Binary Operator Overload

**Overloading <<**

```cpp
inline ostream& operator<<(ostream &os, Number &aNum)
{
    os << -aNum;
    return os;
}
```

- As with most binary operators, << must be overloaded globally.
- It takes an output stream reference (ostream &) as first argument.
- It takes a reference to whatever type you wish to overload the operator for as the second argument
- You need to return an ostream reference (ostream &) which is usually going to be the first parameter.
  - Allows chaining, such as cout << num1 << ", " << num2;

**Overloading >>**

```cpp
inline istream& operator>>(istream &is, Number &aNum)
{
    int value;
    is >> value;
    aNum.setValue(value);
    return is;
}
```

- Overloading the >> operator is a little trickier because you either need to use >> again to actually get input OR you can use lower level routines to access the character stream directly.
- For this simple definition of operator>>, the easier method works.
- We'll cover some cases later in the semester where you need to drop down to the lower level method.

**Consequences of Overloading Globally**

- Whenever we overload globally instead of in the context of a particular class, the overload is implemented "outside of" that class.
  - Private members are inaccessible
- Before you get tempted to make more member variables public to get around this, C++ has a mechanism to make exceptions to the "private" designation.
- It's called a "friend" function

```cpp
class Number
{
    public:
        friend ostream& operator<<(ostream &os, Number &aNum);
        friend istream& operator>>(istream &is, Number &aNum);
        friend int operator+ (const Number &n1,const Number &n2);
    // rest of definition here...
}
```

Demonstration #6

Overloaded <<,>>

Lecture 5

Final Thoughts…