Welcome

Object Oriented Programming with C++
CIS 265

Week 4 – C with Classes

Christopher K. Burns

Schedule

Week		Content
1	1/11	*Review* Chapter 1 Intro to Computers and C++ Programming Chapter 2 Control Structures Chapter 3 Functions Chapter 4 Arrays
2	1/18	Additional Array and Function Topics Chapter 5 Pointers and Strings Lab 1 – Functions, Arrays, and Strings Homework 1 Assigned
3	1/25	Chapter 6 Classes and Data Abstraction
4	2/1	Chapter 7 Classes: Part I Lab 2 – Classes 1 Homework 1 Due
5	2/8	***MID-TERM EXAMINATION*** *(Chapters 1 through 7*)* **only portions of chapter 7 that were covered in class* *Final Project Assigned*
6	2/15	Chapter 7 Classes: Part II Chapter 8 Operator Overloading Homework 2 Assigned
7	2/22	Chapter 8 Operator Overloading Homework 2 Due Lab 3 – Classes 2
8	3/1	Chapter 9 Inheritance: Part I Homework 3 Assigned
9	3/8	Chapter 9 Inheritance: Part II Chapter 10 Polymorphism Homework 3 Due
10	3/15	Chapter 10 Polymorphism Lab 4 – Inheritance and Polymorphism
11	3/22	***FINAL PROJECTS DUE***

Agenda

Tonight's agenda

- Review
  - Pointers and strings
  - Structs and Classes

- Accessor methods
- Constructors and Desctructors
- Static members
- this pointer

- LAB 2

Object Oriented Programming

Home Work

Study for Mid-Term

Finish Lab 2 if not done in class

Mid Term

Mid Term will cover lecture material as well as text chapters 1-6

Mid Term will be a combination of multiple choice and a short essay

Mid Term will be open book/notes

Mid Term review will occur before exam

Looking at pointers and references

What value would i contain after the following lines of code executed?

```
int  x = 3;
int& y =  x;
int* z = &y;
x++;
y -= 2;
*z += 1;
```

Looking at pointers and references

Character array: char *a*[10];

a can act like a pointer

a can be passed as a char*

a is equivalent to &a[0] (address of first character)

a is a pointer to the first character of the array, but it is a constant pointer:

```
char a[10];
char* b;
b = a; // this okay
a = b; // can't do this because address of a is constant
```

Declaring a Struct

```
struct Car
{
  float odometerReading;
  float blueBookValue;
  long  lastOilChangeDate;
  float lifeTimeServiceCost;
  float originalPrice;
  char  ownerName[MAX_STRING_LEN];
};
```

Struct

Passing a struct to a function by value

```
void printMileage( Car car )
{
  cout << "Meleage is " << car.odometerReading << endl;
}

main()
{
  Car myCar;
  myCar.odometerReading = 20000;
  printMileage( myCar );
}
```

Struct

Passing a struct to a function as a pointer

(because some structs can get quite large, it is preferable to pass them as pointers)

```
void printMileage( Car* car )
{
  cout << "Meleage is " << car->odometerReading << endl;
}

main()
{
  Car myCar;
  myCar.odometerReading = 20000;
  printMileage( &myCar );
}
```

Struct

Passing a struct to a function by reference

(because some structs can get quite large, it is preferable to pass them as pointers)

```
void printMileage( Car& car )
{
  cout << "Meleage is " << car.odometerReading << endl;
}

main()
{
  Car myCar;
  myCar.odometerReading = 20000;
  printMileage( myCar );
}
```

Struct becomes a Class

```
#include <iostream>
using namespace std;

const int MAX_STRING_LEN = 128;

class Car
{
public:
  float odometerReading;
  float lifeTimeServiceCost;
  float originalPrice;
  char  ownerName[MAX_STRING_LEN];
  // a function in our struct:
  bool isCarALemon()
  {
    return ( lifeTimeServiceCost > originalPrice );
  }
};


void main()
{
  Car aCar;

}
```

Classes

Programmers like to keep their class definitions clean so that they are easy to read.

This includes implementing functions outside of the class definition.

This is done by prototyping the function in the class, and implementing it outside.

The outside implementation is done by “scoping” (the :: operator) the function name within the class name:

```
class ClassName
{
  public:
    int someIntVariable;
    bool methodName( int anIntValue ); // this is just a
                                       //  regular protoype
};

// implement our function outside the class definition
bool ClassName::methodName( int anIntValue )
{
   // do something meaningful
   return true;
}
```

Separating Interface from Implementation

```
class Car
{
  public:
    float odometerReading;
    float lifeTimeServiceCost;
    float originalPrice;
    bool isCarALemon(); // this is now just a prototype
};

bool Car::isCarALemon()
{
   return ( lifeTimeServiceCost > originalPrice );
}
```

Classes - Data Protection

You should make your instance variables all private.

If a consumer of the class needs access to these private variables, accessor methods can be used.

Accessor methods allow the class consumer to get the value of a private member and/or set the value.

A read-only member can be implemented by only having a get accessor and not a set accessor method.

Classes - Data Protection

The set, is and get methods follow a human readable ordering of verb-noun

setVariable
getVariable
isVariable

When defining your own methods, try to use this convention

Classes - Data Protection

```
#include <iostream>
using namespace std;

class Student
{
private:
  long id;

public:
  long getID();
  void setID( long );
};

long Student::getID()
{
  return id;
}

void Student::setID( long inID )
{
  id = inID;
}
```

```
void main()
{
  Student student;

  student.setID( 1 );

  cout << " id:     " <<
   student.getID() << endl;
}
```

Classes - Data Protection

The *is* accessor prefix is primarily for boolean class properties.

Example – isEnrolled()

```
class Student
{
private:
  bool enrolled;
public:
  bool isEnrolled();
  void setEnrolled( bool );
};
```

Classes - Data Protection

get and set methods for simple data types such as long and floats is straight forward to implement.

```
type getVariable()
{
  return variable;
}
```

```
void setVariable( type in )
{
  variable = in;
}
```

It requires a little more work for more complex data types such as structs, arrays and classes.

Classes - Data Protection

When you implement a set method for an array, you need to copy all the input elements into your private variable.

When you implement a get method for an array, you'll most likely want to return a pointer to the private variable.

- Only return *const* pointers to private variables.
- The const type ensures that the caller cannot modify the contents of the private variable directly.

Classes - Data Protection

get and set for Strings

```
const char* getString()
{
  return string;
}

void setString( char* inString )
{
  strcpy( string, inString );
}
```

Classes - Data Protection

```
#include <iostream>
using namespace std;

const int MAX_STRING_LENGTH = 256;

class Student
{
private:
  char name[ MAX_STRING_LENGTH ];
  long id;

public:
  const char* getName();
  void setName( char* );
  long getID();
  void setID( long );
};

const char* Student::getName()
{
  return name;
}

void Student::setName( char* inName )
{
  strcpy( name, inName );
}
```

```
long Student::getID()
{
  return id;
}

void Student::setID( long inID )
{
  id = inID;
}

void main()
{
  Student student;

  student.setID( 1 );
  student.setName( "Stan Marsh" );

  cout << "name:" << student.getName() << endl;
  cout << "id:  " << student.getID() << endl;
}
```

Class Constructors and Destructors

All classes have constructor and a destructor methods.

If they are not defined, default constructors and destructors are generated by the compiler.

A constructor is called when the class is created, and destructor when it is destroyed.

A constructor can perform initialization operations when class is created

A destructor can perform clean-up operations when class is to be destroyed

Class Constructors and Desctructors

The constructor of a class has the same name as the class.
e.g. constructor method for Car would be Car()

Destructor has the same name as its class preceded by a tilde. E.g. for Car it would be ~Car()

Neither the constructor, nor destructor return values.

Constructors can have arguments, the default constructor does not.

Destructors cannot have arguments.

Default Constructor

```
#include <iostream>
using namespace std;

const int MAX_STRING_LENGTH = 256;

class Student
{
private:
  char name[ MAX_STRING_LENGTH ];
  long id;
  bool enrolled;

public:
  Student(); // default constructor
};

Student::Student() // default constructor implementation
{
  strcpy( name, "not defined" );
  id = 0;
  enrolled = false;
}
```

Overloading the constructor

The constructor is a special type of method call, and can be overloaded.

Overloading the constructor can simplify initialization of members.

If you have overloaded constructors, but did not implement a default constructor, the default constructor will not exist.

When is the constructor called?

```
#include <iostream>
using namespace std;

class Student
{
private:
  long id;

public:
  Student();        // default constructor
  Student( int );   // overloaded constructor
  ~Student();       // destructor
};

Student::Student()
{
  id = 0;
  cout << "default constructor called" << endl;
}

Student::Student( int inID )
{
  id = inID;
  cout << "overloaded constructor called" << endl;
}
```

```
Student::~Student()
{
  cout << "destructor called ";
  cout << "for student: " << id << endl;
}

void main()
{
  Student student1;
  Student student2( 1 );
  cout << "doing stuff" << endl;
}
```

```
default constructor called
overloaded constructor called
doing stuff
destructor called for student: 1
destructor called for student: 0
```

Default Parameters

Functions can have default parameters.

A parameter with a default value is “optional”

Only right most parameters can have defaults. That is to say if your first parameter has a default, so must your second, and so on.

Default Parameters

```
#include <iostream>
using namespace std;

// function with default parameters
//  only right most parameters can have defaults
int someFunction( int x, int y = 1, int z = 1 );

void main()
{
  someFunction( 5 );
  someFunction( 2, 2 );
  //someFunction( 1, , 5 ); // cannot skip params
}

int someFunction( int x, int y, int z )
{
  cout << " x = " << x;
  cout << " y = " << y;
  cout << " z = " << z << endl;
  return x * y * z;
}
```

Default Parameters

It can sometimes be helpful to include default parameters with overloaded constructors

```
class Student
{
private:
  long id;

public:
  Student( int = 0 );  // overloaded constructor
  ~Student();          // destructor
};
```

Static Members

Class vs. Object

Your class can exist and be called directly without having any instances. These are class level methods, and called static.

An instance of you class is an object. Your class may include instance data that is only accessible to objects.

Static Methods

```
#include <iostream>
using namespace std;

class Student
{
private:
  long id;
public:
  static print()
  {
    cout << "Student class" << endl;
  }
  long getID() { return id; }
  void setID( long inID ) { id = inID; }
};

void main()
{
  Student::print();
  Student studentObject;
  studentObject.setID( 1 );
}
```

Initializing Static Members

```
#include <iostream>
using namespace std;
const int MAX_STRING_LENGTH = 256;

class Student
{
private:
  long id;
  static char className[];
public:
  static print()
  {
    cout << className << endl;
  }
  long getID() { return id; }
  void setID( long inID ) { id = inID; }
};

char Student::className[] = "Student Class"; // static members declared at file scope

void main()
{
  Student::print();
  Student studentObject;
  studentObject.setID( 1 );
}
```

Static Members Example

```
#include <iostream>
using namespace std;
const int MAX_STRING_LENGTH = 256;

class Student
{
private:
  long id;
  static long studentCount;
public:
  Student()
  {
    studentCount++;
  }
  static long getStudentCount()
  {
    return studentCount;
  }
  long getID() { return id; }
  void setID( long inID ) { id = inID; }
};

long Student::studentCount = 0;
```

```
void main()
{
  Student student1;
  Student student2;
  Student student3;
  cout << "There are " <<
    Student::getStudentCount();
  cout << " students" << endl;
}
```

```
There are 3 students
```

Looking at thyself – this pointer

Every class instance, object, has a special private member called the *this* pointer.

The *this* pointer refers to the instance.

It allows the class instance to know who it is, as well as being able to tell others who it is.

The *this* pointer

There are many uses for the *this* pointer, three of which are:

1. Avoiding name collisions
2. Cascading method calls
3. Passing an instance of yourself

The *this* pointer

Example 1 – get/set – avoiding name collisions

```
class Student
{
private:
  long id;
public:
  long getID() { return id; }
  void setID( long id )
  {
    this->id = id;
  }
};
```

The *this* pointer

Example 2 – cascading method calls

```
class Student
{
private:
  long id;
  bool enrolled;
public:
  long getID() { return id; }
  Student& setID( long id )
    { this->id = id; return *this; }
  bool isEnrolled() { return enrolled; }
  Student& setEnrolled( bool enrolled )
    { this->enrolled = enrolled; return *this; }
};

void main()
{
  Student student;
  student.setEnrolled( true ).setID( 1 );
}
```

The *this* pointer

Example 3 – passing your instance around

```
#include <iostream>
using namespace std;

// constants
const int MAX_STRING_LENGTH = 256;
const int MAX_STUDENTS = 20;

// class Student interface
class Student
{
private:
  static long studentCount;
  static Student* studentList[ MAX_STUDENTS ];

  char name[ MAX_STRING_LENGTH ];
  bool enrolled;
  long id;

public:
  Student( char*, bool );

  const char* getName();
  void setName( char* );
  bool isEnrolled();
  void setEnrolled( bool );
  long getID();

  static Student* getStudentFromID( long id );
  static long getStudentCount();
};
```

```
// class Student implmentation
Student::Student( char* name, bool enrolled )
{
  setName( name );
  setEnrolled( enrolled );
  id = studentCount;
  studentList[ Student::studentCount ] = this;
  studentCount++;
}

const char* Student::getName()
{
  return name;
}

void Student::setName( char* name )
{
  strcpy( this->name, name );
}

bool Student::isEnrolled()
{
  return enrolled;
}

void Student::setEnrolled( bool enrolled )
{
  this->enrolled = enrolled;
}
```

The *this* pointer

Example 3 – passing your instance around – cont'd

```
long Student::getID()
{
  return id;
}

// class Student static method implmentation
Student* Student::getStudentFromID( long id )
{
  if ( ( id <= studentCount ) && ( id >= 0 ) )
  {
    return studentList[ id ];
  }
  else
  {
    return 0;
  }
}

long Student::getStudentCount()
{
  return studentCount;
}

// class Student static member initialization
long Student::studentCount = 0;
Student* Student::studentList[ MAX_STUDENTS ];
```

```
// main routine to demonstrate Student
void main()
{
  // get students
  do
  {
    char name[ MAX_STRING_LENGTH ];
    bool enrolled;
    cout << "Student name:";
    cin >> name;
    if ( stricmp( name, "end" ) == 0 )
      break;
    cout << "   enrolled?:";
    cin >> enrolled;
    new Student( name, enrolled );
  }
  while( true );
```

The *this* pointer

Example 3 – passing your instance around – cont'd

```
  // print list of students
  for( int i = 0; i < Student::getStudentCount(); i++ )
  {
    cout << "Student Record" << endl;
    cout << " name:    " << Student::getStudentFromID(i)->getName() << endl;
    cout << " id:      " << Student::getStudentFromID(i)->getID() << endl;
    cout << " enrolled:" << Student::getStudentFromID(i)->isEnrolled() << endl;
  }
}
```

Output

```
Student name:Eric
   enrolled?:1
Student name:Kenny
   enrolled?:0
Student name:end
Student Record
 name:    Eric
 id:      0
 enrolled:1
Student Record
 name:    Kenny
 id:      1
 enrolled:0
```

Lab 2

Homework 2 will depend on Lab 2, so keep a copy of it handy