# MECH 215

Lecture 6

Classes

# Classes

- Classes are the foundation of object-oriented programming in the C++
- language.
- A C++ class is a collection of DATA MEMBERS (which may be of different
- types such as int, char, float, int*, etc.)  and MEMBER FUNCTIONS (which
- are given the special name of METHODS).
- Previously data and operations were kept separate, this is called procedural programming.

# Classes

- The class data type allows for the concept of ENCAPSULAT toION - a grouping of data members (those characteristics which are defined using data types) and the functions (those characteristics which are derived from data members) these two elements are combined operate on this data into a unified "object".
- The object has both data types and operations that may be either given as input to the object or derived by input values.
- The syntax of a class is similar to that of a structure.

# Classes

- The class construct allows for information hiding - a class can restrict access to either its data members or its methods.
  - Information within a class can be hidden by declaring the information to be 'private'.
  - Conversely, information may be visible to users of the class by declaring it to be 'public'.
- Most often the member methods (functions) provide the information needed from the object to other parts of the program.

# Classes

- We will look at the concepts using a simple example based on a known geometrical object.
- A circle can be defined using the radius. From the radius the perimeter and area of a circle can be found.
- In this manner of thinking, if we define a class circle with a member to be the radius, we can define the perimeter and area of the circle as class methods.
- We will look at an example of the code in the following slides.

# Classes

- const float pi=3.1415926; // Need this for calculations
- #include<iostream>
- #include<string>
- using namespace std;
- class circle  // This will tell compiler we are creating a class called circle
- {
- float radius;  // We declare a (private) member data type floating point called radius
- public:  // We declare public member methods (public means other parts can use)
- float area();  // Floating point return from method area
- float circumference(); // Floating point return from method area
- void setradius(float r); // Void method used as input radius
- };

# Classes

- float circle::area()  //Now define what area circle does note :: used for methods
- {
- return (pi*radius*radius);  //Area is pi time radius squared
- }
- float circle::circumference() //Now define what circum. does
- {
- return (2*pi*radius);  //Perimeter is 2*pi*radius
- }
- void circle::setradius(float r) //Define what setradius does
- {
- radius=r;  //Input is set to radius and used for other functions
- }

# Classes

- int main()  //This program will use the classes to perform operations
- {
- circle big,small;
- big.setradius(10.00);
- cout << "The big circle has an area of " << big.area()<<endl;
- cout <<"The big circle has a perimeter of " <<big.circumference()<<endl;
- small.setradius(1.00);
- cout <<" The small circle has an area of " <<small.area()<<endl;
- cout <<"The small circle has a perimeter of " << small.circumference()<<endl;
- return 0;
- }

# Classes

- The output of the code once it has been run is:
  - The big circle has an area of 314.159
  - The big circle has a perimeter of 62.8318
  - The small circle has an area of 3.14159
  - The small circle has a perimeter of 6.28319
- Note that the class does not need additional input since the setradius method provides all of the information required.
- When we define the radius of the circle class, all of the other things will be obtained from that input.

# Classes

- Next, we can have another file containing the implementation of the methods found in the class . This file can be called circle_methods.C :
- This will be effectively the same as the class we defined previously. Where all of the methods will be defined and where we will also set the default values.
- This is called the default constructor (we will talk more about constructors in the next slide).

# Classes

- DEFAULT CONSTRUCTORS
- If a constructor takes no arguments, it is known as a default constructor.

- OVERLOADED CONSTRUCTORS
- A class definition may contain more than one definition of a constructor, we may define different versions of the constructor function by OVERLOADING the constructor (that is multiple definitions which vary in their argument list).

# Classes

- CONSTRUCTORS
- The above circle class contains a method called “SetRadius” which must be manually invoked by the user to set the radius of a certain circle object to a specific value.  This is burdensome and error-prone, a user of the class may forget to do so.
- C++ has a way of automatically initializing the data members of a class. The method makes use of CONSTRUCTORS .  Constructors are special purpose methods (which have the same name as the class), constructors are automatically called by the compiler when an object of the class is defined.
- Constructors differ from ordinary functions and methods in that NO RETURN TYPE (NOT EVEN A RETURN TYPE OF VOID) is specified in the definition of a constructor.

# Classes

- We can also write the class using inline format.
- class circle
- {
- float radius; // Private member
- public:
- float area(){return (pi*radius*radius);}; // Same instructions one line
- float circumference(){return (2*pi*radius);};
- void setradius(float r){radius=r;}
- };

# Classes

- IT IS AN ERROR TO ATTEMPT TO DIRECTLY ACCESS THE PRIVATE DATA OF A  CLASS.
- The compiler will not let you. This is why the members are called “private”.
- You may access public members and methods. Make sure that you only try to do so with your program.

# Classes

- Once you have made a class you would want it to be generally available rather than writing it all of the time you can define a library which would allow any function that you want to have access to your class.
- Just as we used header files to include function prototypes, we may make use of header files to include class definitions. The following example is our class Circle example redone using header files.
- We also make use of the #ifndef compiler directive to prevent multiple redefinitions of identifiers in cases where a header file is included more than one time:

# Classes

- We can put the definition of the class Circle in a file called circle.h (any file name may be used... it is good to follow the class_name.h convention): all this goes into the file called circle.h.
- #ifndef circle_h
- #define circle_h
- const float pi = 3.1415926;
- class Circle
- {
- float radius; // everything in a class is private by default
- public: // we want the following methods to be public, so we
- Circle(); // default constructor define constructor later.
- Circle(float r); // overloaded constructor
- float Area();
- float Circumference();
- float GetRadius();
- }; // NOTE: THIS LAST SEMICOLON IS REQUIRED !!!!
- #endif

# Classes

- #include "circle.h"  // this will read the file circle.h which contains the class declarations (what is in the class)
- #include <iostream>
- using namespace std;
- float Circle::Area()  // Now we actually define the methods
- {
- return ( pi * radius * radius) ;
- }
- float Circle::Circumference()
- {
- return ( 2 * pi * radius) ;
- }
- float  Circle::GetRadius()
- {
- return radius;
- }  //Continued next slide

# Classes

- Circle::Circle()
- {
- cout << "Default constructor called... " << endl; //Define default constructor
- cout << "Setting circle's radius to 1.00 " << endl;
- radius = 1.00 ;
- }
- Circle::Circle(float r)
- {
- cout << "Overloaded constructor invoked ... " << endl;
- radius = r;
- }

# Classes

- We can then use this class in any program we wish by linking the objects in the compile.
- #include "circle.h"
- #include <iostream>
- using namespace std;
- int main()
- {
- Circle circle_a, circle_b(456.89);
- cout << "Circle A has radius " << circle_a.GetRadius() << endl;
- cout << "Circle B has radius " <<  circle_b.GetRadius() << endl;
- return 0;
- }

# Classes

- Compile using
- g++ -o circle_main circle_methods.c circle_main.cp
- When we run the program we get:
- [grace] [/home/d/don/Mech215] > ./circle_main
- Default constructor called...
- Setting circle's radius to 1.00
- Overloaded constructor invoked ...
- Circle A has radius 1
- Circle B has radius 456.89

# Classes

- We could also obtain information about the other properties of the circle by selecting the appropriate method.
- And printing or using the returned value from the method.

# Classes

- Operator Overloading
- C++ allows to overload built-in operators such as +, - , *, etc.
- Let us overload the + operator such that it will "add" two circle objects together.  For the purposes of our example, we will define the addition of two circles to give a third circle  whose radius is the sum of the two circles being added together.

# Classes Declare

- const float pi = 3.1415926;
- #include <iostream>
- #include <string>
- using namespace std;
- class Circle
- {
- float radius;  // Available to the class
- public:  // Available to all
- Circle();  // default constructor
- Circle(float r); // overloaded constructor
- float Area();
- float Circumference();
- float GetRadius();
- Circle operator+(Circle);
- };

# Classes Methods

- float Circle::Area()
- {
- return ( pi * radius * radius) ;
- }
- float Circle::Circumference()
- {
- return ( 2 * pi * radius) ;
- }
- float  Circle::GetRadius()
- {
- return radius;
- }

# Classes Constructors

- Circle::Circle()
- {
- cout << "Default constructor called... " << endl;
- cout << "Setting circle's radius to 1.00 " << endl;
- radius = 1.00 ;
- }
- Circle::Circle(float r)
- {
- cout << "Overloaded constructor invoked ... " << endl;
- radius = r;
- }

# Classes Operator Overloading

- Circle Circle::operator+(Circle a_circle)
- {
- Circle temp ; // note default constructor will be called on temp
- temp.radius = radius + a_circle.radius;
- return temp;
- }

# Classes Main

- int main()
- {
- // create two Circle "objects"
- Circle A, B(456.89);
- Circle C;
- cout << "Circle A has radius " << A.GetRadius() << endl;
- cout << "Circle B has radius " <<  B.GetRadius() << endl;
- // add A and B to get C
- C = A + B ; // Radius of C  should be 1.00 + 456.89
- cout << "Circle C has radius " << C.GetRadius() << endl;
- return 0;
- }

# Classes Output

- Default constructor called...
- Setting circle's radius to 1.00
- Overloaded constructor invoked ...
- Default constructor called...
- Setting circle's radius to 1.00
- Circle A has radius 1
- Circle B has radius 456.89
- Default constructor called...
- Setting circle's radius to 1.00
- Circle C has radius 457.89

# Classes Overloaded Operators

- The line C = A + B ; can also be written as: C = A.operator+(B);

- Written in this manner, it is clearer that we are actually invoking the method called operator+ belonging to the circle A and passing to this method the circle B as a parameter. The return value of the operator+ method is then assigned to the circle C.
- If we write it as C = A + B, it is more apparent that we are "adding" two circles together and assigning the result of the "addition“ to another circle.

# Classes Destructors

- Constructors are used to perform INITIALIZATION of data members. This initialization may  involve some DYNAMIC MEMORY allocation (for example assign some pointer an address returned by the new operator).
- Recall that when an object which dynamically allocates some memory goes out of scope, the memory used for that object's data members are returned to the operating system (and given back to the heap space0, HOWEVER ANY DYNAMICALLY ALLOCATED MEMORY IS NOT RETURNED.
- Thus, a memory leak is said to occur, and we have the possibility of exhausting the heap space causing the program to crash.

# Classes Destructor

- There are two solutions. One is to have the user of the class, explicitly give back any dynamically allocate memory with the delete operator when that memory is no longer needed: E.g.
- void  Big::clear()
- {
- delete [] string;
- }

# Classes Destructor

- There is a better way which uses DESTRUCTORS.
- DESTRUCTOR functions are special functions inside of class objects, they NEITHER TAKE ANY ARGUMENTS, NOR DO THEY RETURN A VALUE; DESTRUCTORS ARE HIGHLY SPECIALIZED FUNCTIONSWHICH EXIST ONLY TO RELEASE ANY MEMORY WHICH WAS DYNAMICALLY ALLOCATED BY A CONSTRUCTOR!!!!!!
- A DESTRUCTOR's name is similar to the constructor in the respect that each is the same as the class name. However, a destructor name is preceded with the tilde (~) symbol.
- Here is a program which makes use of the destructor ~Big() written by T. Obuchowitcz.

# Classes Destructor

- #include <iostream.h> //use iostream
- #include <stdlib.h>  //use cstdlib also use cstring.
- const long OneMillion = 1000000;
- class Big
- {
- char* string ; // a data member which is a pointer to a character..i.e a string
- public:
- Big(); // default constructor which will do some DYNAMIC memory allocation
- ~Big(); // destructor which will de-allocate dynamic memory
- };

# Classes Destructor

- Big::Big()
- {
- string = new char [OneMillion];  // get 1 000 000 char cells
- if (!string)
- {
- cout << "OUT OF MEMORY!!!!" << endl;
- exit(1);
- }
- }
- Big::~Big()
- {
- delete [] string;
- }

# Classes Destructor

```
void garbage(void)
{
Big oops;
// do some local processing of object oops
}
void main(void)
{
int i;
for(i = 0; i < 10000; i++)
{
  cout << "I = " << i << endl;
  garbage();
}
}
```

# Classes Destructor

- This program completes its for loop with no memory problems:

- I = 9988
- I = 9989
- I = 9990
- I = 9991
- I = 9992
- I = 9993
- I = 9994
- I = 9995
- I = 9996
- I = 9997
- I = 9998
- I = 9999

# Classes

- Without freeing memory the program would eventually use up all free memory:
- My output is (for example on a sun system):

- I = 223
- I = 224
- I = 225
- I = 226
- I = 227
- I = 228
- I = 229
- Abort

# Classes

- DIFFERENCES BETWEEN A C++ STRUCT AND A C++ CLASS
- A C++ struct  is very similar to a C++ class.  In fact , the only difference between C++ structs and classes is that in A STRUCT EVERYTHING IS CONSIDERED PUBLIC (unless explicitly denoted as private) BY DEFAULT AND IN A CLASS EVERYTHING IS CONSIDERED PRIVATE (unless explicitly denoted as public by use of the public keyword).
- Traditionally, C++ programmers use the class construct when we want to encapsulate the data members together with the methods which operate on these members. A struct is traditionally used only when there are data members (all public) and no methods.  It is possible to do otherwise as we shall see in the following example.

# Struct as class

- // Author: Ted Obuchowicz
- //  Oct. 29, 2002
- // file: fraction_struct_class.C
- #include <iostream>
- #include <string>
- using namespace std;
- struct fraction
- {
- private: // in a struct everything is public unless explicitly  made private
- int numerator, denominator;
- public:
- fraction() {  numerator = 1; denominator = 1; } ; // inline definition of default constructor
- fraction(int top, int bottom ) { numerator = top; denominator = bottom; };
- return 0;
- }

# Struct as class

- fraction  add(fraction x)
- {
- fraction temp;
- temp.numerator = numerator * x.denominator + x.numerator * denominator;
- temp.denominator = denominator * x.denominator;
- return temp;
- };
- void print_fraction()
- {
- cout << numerator << " / " << denominator << endl;
- }
- };

# Struct as class

- int main()
- {
- fraction f1(2,3) ;
- fraction f2(3,4) ;
- fraction sum;
- f1.print_fraction();
- f2.print_fraction();
- sum =  f1.add(f2);
- sum.print_fraction();
- return 0;
- }

# Next Lecture

- In the next lecture we will discuss file I/O.