C++ Lecture 13

* Object-Oriented Programming: Constructors, Abstract Data Types
* CIS 251 • Shelby-Hoover Campus

Class Definition Miscellany

* A class may include a member variable of another class type (similar to a structure hierarchy)
* Member function definitions can be overloaded
  + Example: the DayOfYear class could have another member function named setMonth that takes in a string for the name of the month and converts it to the correct number
  + If the argument to setMonth call is an int, the original version of setMonth is used; if the argument is a string, the second version of setMonth is used

Overloading in a Class

* The class BankAccount with an overloaded setBalance mutator:  
    
  class BankAccount  
  {  
  public:  
   void setBalance(int dollars, int cents);  
   void setBalance(int dollars);  
   void setInterestRate(double rate);  
   void update(); // adds one year of interest  
   double getBalance();  
   double getInterestRate();  
   void output();  
  private:  
   double balance;  
   double interestRate;  
  }; // Don't forget the semicolon here!

Function Definitions (1)

* Both of these mutator member functions are available to a BankAccount object:  
    
  void BankAccount::setBalance(int dollars, int cents)  
  {  
   if ((dollars >= 0) && (cents >= 0))  
   balance = dollars + 0.01 \* cents;  
   else  
   cout << "Illegal arguments for balance." << endl;  
  }  
    
  void BankAccount::setBalance(int dollars)  
  {  
   if (dollars >= 0)  
   balance = dollars;  
   else  
   cout << "Illegal argument for balance." << endl;  
  }

Function Definitions (2)

* These member functions are also available to a BankAccount object:  
    
  void BankAccount::setInterestRate(int rate)  
  {  
   if (rate >= 0)  
   interestRate = rate;  
   else  
   cout << "Illegal argument for rate." << endl;  
  }  
    
  void BankAccount::update()  
  {  
   balance += balance \* interestRate / 100.0;  
  }  
    
  double BankAccount::getBalance()  
  {  
   return balance;  
  }  
    
  double BankAccount::getInterestRate()  
  {  
   return interestRate;  
  }

Function Definitions (3)

* As with the DayOfYear class, the input member function for BankAccount stores the user’s input in temporary variables before sending these to the mutator member functions that perform appropriate validation:  
    
  void BankAccount::input()  
  {  
   int tempDollars, tempCents;  
   double tempRate;  
    
   cout << "Enter the balance as follows." << endl;  
   cout << "Dollars: ";  
   cin >> tempDollars;  
   cout << "Cents: ";  
   cin >> tempCents;  
   setBalance(tempDollars, tempCents);  
    
   cout << "Enter the interest rate: ";  
   cin >> tempRate;  
   setInterestRate(tempRate);  
  }

Function Definitions (4)

* These definitions for input and output work only with the console; in a moment, you’ll see how to define these member functions so that they can communicate either with the console or with a file

void BankAccount::output()  
{  
 cout.setf(ios::fixed);  
 cout.setf(ios::showpoint);  
 cout.precision(2);  
 cout << "Account balance $" << balance << endl;  
 cout << "Interest rate " << interestRate << "%" << endl;  
}

Objects and Functions

* A function may include a value or reference parameter of a class type in its header
* A function may also return an object of a class type (the return type should match the class type)
* An object parameter may be flexible to accept arguments of several related types
  + Example: an object of the generic class ostream can refer to cout or to an ofstream variable (similar to an istream parameter as used in chapter 6)
  + Such a parameter can be used to specify output to a file or to the console

Stream Parameter Example 1

* This definition of output includes a reference parameter of type ostream:  
    
  void DayOfYear::output(ostream& outs)  
  {  
   outs << "month = " << month << ", day = " << day << endl;  
  }
* The argument to this function could be cout:  
    
  examDate.output(cout);
* The argument could also be an ofstream variable:  
    
  ofstream examFile;  
  examFile.open("exams.dat");  
    
  examDate.output(examFile);

Stream Parameter Example 2

* When rewriting an input member function, remove any prompts that would be used only with console input (the prompts should be displayed in the function that calls input):  
    
  void BankAccount::input(istream& ins)  
  {  
   int tempDollars, tempCents;  
   double tempRate;  
    
   ins >> tempDollars;  
   ins >> tempCents;  
   setBalance(tempDollars, tempCents);  
    
   ins >> tempRate;  
   setInterestRate(tempRate);  
  }

Stream Parameter Example 3

* When rewriting an output member function, make sure that every reference to cout is changed to the ostream& parameter name:  
    
  void BankAccount::output(ostream& outs)  
  {  
   outs.setf(ios::fixed);  
   outs.setf(ios::showpoint);  
   outs.precision(2);  
   outs << "Account balance $" << balance << endl;  
   outs << "Interest rate " << interestRate << "%" << endl;  
  }

Constructors

* A programmer may want specific code executed when an object is created
* A **constructor** is a special member function that is automatically invoked upon the creation of an object
  + The name of the constructor matches the class name
  + **There is no return type** (not even void)
  + As many parameters as needed can be included
  + If the definition is written outside the class definition, don’t forget the type qualifier and the scope resolution operator (e.g., ClassName::ClassName(parameters))
  + Most constructors assign a value to every member variable, even to those for which a parameter is not provided

Constructor Example

* This prototype for a constructor for the BankAccount class would be placed in the class’s public section:  
    
  BankAccount(int dollars, int cents, double rate);
* This is the definition for the constructor as it would appear outside of the class definition:  
    
  BankAccount::BankAccount(int dollars, int cents, double rate)  
  {  
   if ((dollars < 0) || (cents < 0) || (rate < 0))  
   {  
   cout << "Illegal values for money or interest rate.\n";  
   cout << "Setting both values to zero.\n";  
   balance = 0.0;  
   interestRate = 0.0;  
   }  
   else  
   {  
   balance = dollars + 0.01 \* cents;  
   interestRate = rate;  
   }  
  }

Using Constructors

* Invoke a constructor with parameters by placing the arguments to those parameters in parentheses after the object variable name:  
    
  BankAccount account1(100, 0, 2.3);  
  + balance = 100 + 0.01 \* 0;
  + interestRate = 2.3:

Constructor Overloading

* A class may contain multiple constructors to allow a programmer to construct an object in different ways (for example, another constructor that accepts the dollars and the rate but does not require the cents):  
    
  BankAccount::BankAccount(int dollars, double rate)  
  {  
   if ((dollars < 0) || (rate < 0))  
   {  
   cout << "Illegal values for money or interest rate.\n";  
   cout << "Setting both values to zero.\n";  
   balance = 0.0;  
   interestRate = 0.0;  
   }  
   else  
   {  
   balance = dollars;  
   interestRate = rate;  
   }  
  }

The Default Constructor

* If a class does not define a constructor, C++ provides a **default constructor** that does not require arguments
* If you define any constructor, you should include one that does not require arguments in place of the default constructor; it should assign default values to the member variables
* To invoke the default (or no-argument) constructor, leave off the parentheses after the object variable name

Initialization Section

* A constructor’s header may contain special syntax known as an **initialization section** to assign initial values to the member variables
  + The section begins after the parameter list with a colon
  + Each member variable to be initialized is listed after the colon, with the value to be assigned in parentheses after the variable name
  + Multiple variables are separated by commas
* Variables initialized in the initialization section do not require an initialization statement inside the body

Initialization Example

* A constructor with no parameters for BankAccount objects:  
    
  BankAccount::BankAccount() : balance(0), interestRate(0.0)  
  {  
   // body is empty  
  }
* Invoking this constructor:  
    
  BankAccount account2; // no parentheses  
  + balance = 0;
  + interestRate = 0.0;

Code with Constructors

* A program may call a constructor after a variable has been declared using an assignment statement:  
    
  existingVariable = ClassName(arguments);
* Example: a main function that uses three different constructors:  
    
  int main()  
  {  
   BankAccount account1(100, 2.3), account2;  
   cout << "account1 initial values:" << endl;  
   account1.output(cout);  
   cout << "account2 initial values:" << endl;  
   account2.output(cout);  
    
   // This line assigns a new object to account1  
   account1 = BankAccount(999, 99, 5.5);  
   cout << "account1 new values:" << endl;  
   account1.output(cout);  
   return 0;  
  }

Abstract Data Types (ADT)

* A programmer has great flexibility in writing a class (public vs. private, what member functions to provide, what level of access is granted to specific external functions)
* Usually it’s best to strive to write a class as an **abstract data type** (ADT) so that a programmer who knows how to use objects of the class does not know how the class works behind the scenes
  + The **interface** of the class includes the details a programmer needs in order to use the class (prototypes of the public member functions, comments describing how to use them)
  + The **implementation** of the class is the code that the programmer using the class is not aware of (private members, definitions of public member functions)
  + A class’s implementation may change without the interface changing, and the programmer wouldn’t know the difference

Guidelines for ADTs

* Make all the member variables private members of the class
* Make each of the basic operations a public member of the class, with full specifications on how to use them
* Make any helper functions private members

Separate Compilation

* C++ allows a programmer to place the class definition and member function and friend function definitions in separate files
  + The **interface file** contains the class definition; its file extension is usually .h instead of .cpp
  + The **implementation file** contains the definitions of the member functions and friend functions (chapter 11)
  + The file in which objects of the class type are declared and used is known as the **application file** or **driver file**
* Both the implementation file and the file for the program in which the class is used should have a special include statement for the interface file, with the file name in quotation marks instead of angle brackets:  
    
   #include "ClassName.h"
* It’s impossible to hide the private members in the interface file, but this is as close to an ADT as C++ allows

Inheritance

* C++ allows a class definition to **inherit** the members of another class
* In an inheritance relationship, the **derived class** (also known as the **child**) receives all of the members of the **base class** (also known as the **parent**) and adds its own
* The derived class cannot access the inherited private members directly; it must go through the base class’s public member functions
* The simplest form of inheritance assumes that all public members of the base class will remain public in the derived class:  
    
  class SavingsAccount : public BankAccount
* Multiple classes may be derived from a single base class