C++ Lecture 12

Grouping Variables

Structures

* Inherited from the C language, a **structure** allows a programmer to define a type that incorporates several variables of varying types, each with its own name
  + The name of the new structure type is known as a **structure tag**
  + The names of the variables that make up the structure definition are **member names**, and the variables themselves are known as **member variables**
* An unlimited number of **structure variables** can be declared from a single structure type
  + The value stored in a structure variable is known as a **structure value**
  + The values of the individual member variables are called **member values**

Hierarchical Structures

* Each structure definition has its own scope; if a program contains multiple structure definitions, the member names can be the same without conflicts

Initialization / Assignment

* A program may use an initialization list similar to an array when declaring a structure variable; the values in the list will be assigned in the order in which the member variables are listed in the definition:  
    
  CDAccount oneYear = { 1500.00, 2.05, 12 };

Structures and Functions

* A function can have a structure variable as a parameter (pass by value or by reference):  
    
  void getData(CDAccount& theAccount)  
  {  
  }
* A function may also return a structure variable (the return type of the function should match the structure tag)

Limitations of Structures

* Structure definitions only include variables, not functions that can be used to modify the variables
* A programmer cannot control how member variables are used (checking to see if a value to be assigned to a member variable is valid before accepting it)
* Common operations must be performed on individual member variables, not structure variables in entirety

Classes

* A **class** is a data type that includes both member variables and member functions
  + Variables declared from class types are known as **objects**
  + Member variables are known as **attributes** or **fields**
  + Member functions are known as **methods**
  + The process of combining several elements into a single package is known as **encapsulation**
* The concept of combining the data for an entity with the operations that can be performed on that data is known as **object-oriented programming**

Member Function Definitions

* A member function definition written outside of the class definition must include a **type qualifier** and the **scope resolution operator** in its header

Declaring, Using Objects

* Since each object has its own values of the member variables, a call to a public member function uses the member variables of the **calling object**

public vs. private

* A private member function may be included to serve as a “helper method” to another member function, but it cannot be invoked by outside code
* If a programmer omits the private and public keywords, C++ assumes that you all members should be private

Accessors and Mutators

* An **accessor** member function (name begins with “get”) allows the user to access a value from within the object
* A **mutator** member function (name begins with “set”) allows the user to modify one or more values in an object

C++ Lecture 13

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

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

The Default Constructor

Abstract Data Types (ADT)

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**

#include "ClassName.h"

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

C++ Lecture 14

External Functions

* Functions that are not a part of a class normally are not allowed direct access to the private members of that class
* Usually a function that deals with two or more objects given equal standing is not made a member function
* Because such a function must interact with the object through public methods, the syntax can become cumbersome:  
    
  bool equal(DayOfYear date1, DayOfYear date2)  
  {  
   return ((date1.get\_month() == date2.get\_month()) &&  
   (date1.get\_day() == date2.get\_day()));  
  }

Friend Functions

* The prototype may be listed in either the private or the public section of the class definition, but it will be public regardless, so the public section is more appropriate

Friends and Members

* Accessors and mutators are still important components of class definitions, as they provide controlled interaction with the private members of the class

Constant Parameters

* Pass-by-reference parameters are more efficient than pass-by-value: the function uses the original argument rather than making a duplicate
* To benefit from pass-by-reference without running the risk of damage to the original objects, use a **constant reference parameter**:  
    
  Money add(const Money& amount1, const Money& amount2)  
  {  
   // body remains as on previous slide

Constant Calling Objects

* When an object is used to call a class method, it works similarly to pass-by-reference: the method may make changes to the calling object as needed
* Some methods do not need access to make changes to the calling object
* The keyword const can be added to the end of a method prototype and header to indicate that the method should not be allowed to make changes to the calling object:  
    
  void output(ostream& outs) const;  
    
  void Money::output(ostream& outs) const  
  {  
   // body omitted

Binary Arithmetic Operators

* When an object stores numeric values, a binary arithmetic operator allows a programmer to calculate the sum, difference, product, quotient, or remainder of the values in two objects as a new object
* These functions require two constant reference parameters of the class type to represent the values on either side of the operator
* The return type of each of these functions is the class name (the function returns a new object of that type)
* Within the function, a new object stores the arithmetic result
* The function returns this new object

Comparison Operators

* To compare the values stored in two objects, a class can overload the comparison operators
* These functions require two constant reference parameters of the class type for the objects on either side of the operator
* The return type of each of these functions is bool (the function returns true or false)
* The function compares as many (or as few) of the member variables as needed (a programmer decides what counts as “equality” between two objects)
* The function returns the bool result

Unary Arithmetic Operators

* Remember: a unary operator is one in which there is only one operand (e.g., negation)
* To overload a unary operator, provide a single constant reference parameter of the class type
* The return type of each of these functions is the class name (the function returns a new object of that type)
* The function creates a new object that stores the arithmetic result and returns this new object
* Overloading increment (++) and decrement (--) only provides for the prefix versions of these operators; the postfix version is not covered in the textbook
* #include <iostream>
* #include <fstream>
* #include <string>
* #include <cstdlib>
* #include <algorithm>
* #include <iomanip>
* using namespace std;
* class RetailProp
* {
* public:
* RetailProp();
* string get\_id();
* string get\_tenant();
* int get\_area();
* double get\_rent();
* bool get\_id\_flag();
* bool get\_area\_flag();
* bool get\_rent\_flag();
* void set\_id(string new\_id);
* void set\_tenant(string new\_tenant);
* void set\_area(int new\_area);
* void set\_rent(double new\_rent);
* void set\_id\_flag(bool new\_flag);
* void set\_area\_flag(bool new\_flag);
* void set\_rent\_flag(bool new\_flag);
* double getRentPerSqFoot(double prop\_rent, int unit\_price);
* void input(istream& ins);
* void output(ostream& out);
* private:
* string idNumber;
* string tName;
* int area;
* double rent;
* bool id\_flag;
* bool area\_flag;
* bool rent\_flag;
* };
* int main()
* {
* RetailProp property1, property2, property3, property4, property5;
* ifstream prop;
* prop.open("properties.txt");
* if (prop.fail())
* {
* cout << "Input file opening failed.\n";
* exit(1);
* }
* property1.input(prop);
* property2.input(prop);
* property3.input(prop);
* property4.input(prop);
* property5.input(prop);
* property1.output(cout);
* property2.output(cout);
* property3.output(cout);
* property4.output(cout);
* property5.output(cout);
* prop.close();
* return 0;
* }
* RetailProp::RetailProp()
* {
* idNumber = "??????";
* tName = "vacant";
* area = 0;
* rent = 0;
* id\_flag = false;
* area\_flag = false;
* rent\_flag = false;
* }
* string RetailProp::get\_id()
* {
* return idNumber;
* }
* string RetailProp::get\_tenant()
* {
* return tName;
* }
* int RetailProp::get\_area()
* {
* return area;
* }
* double RetailProp::get\_rent()
* {
* return rent;
* }
* bool RetailProp::get\_id\_flag()
* {
* return id\_flag;
* }
* bool RetailProp::get\_area\_flag()
* {
* return area\_flag;
* }
* bool RetailProp::get\_rent\_flag()
* {
* return rent\_flag;
* }
* void RetailProp::set\_id(string new\_id)
* {
* if (idNumber <= " ")
* idNumber = "??????";
* else
* idNumber = new\_id;
* }
* void RetailProp::set\_tenant(string new\_tenant)
* {
* if (tName <= " ")
* tName = "vacant";
* else
* tName = new\_tenant;
* }
* void RetailProp::set\_area(int new\_area)
* {
* if (area < 0)
* area = 0;
* else
* area = new\_area;
* }
* void RetailProp::set\_rent(double new\_rent)
* {
* if (rent < 0)
* rent = 0;
* else
* rent = new\_rent;
* }
* void RetailProp::set\_id\_flag(bool new\_flag)
* {
* id\_flag = new\_flag;
* }
* void RetailProp::set\_area\_flag(bool new\_flag)
* {
* area\_flag = new\_flag;
* }
* void RetailProp::set\_rent\_flag(bool new\_flag)
* {
* rent\_flag = new\_flag;
* }
* double RetailProp::getRentPerSqFoot(double prop\_rent, int rented\_area)
* {
* return (rent/area);
* }
* void RetailProp::input(istream& ins)
* {
* ins >> idNumber >> tName >> area >> rent;
* if (idNumber.size() > 6)
* {
* set\_id\_flag(true);
* }
* if(area < 50)
* {
* set\_area\_flag(true);
* }
* set\_area(area);
* if(rent < 500)
* {
* set\_rent\_flag(true);
* }
* set\_rent(rent);
* }
* void RetailProp::output(ostream& out)
* {
* cout.setf(ios::fixed);
* cout.setf(ios::showpoint);
* cout.precision(2);
* if (idNumber == "??????")
* {
* out << "Error! Property information not entered.\n";
* cout << endl;
* }
* else
* {
* if(id\_flag == true)
* {
* cout << "Error: Identification Number should not be longer than six characters.\n";
* set\_id\_flag(false);
* }
* cout << "Property #" << idNumber << endl;
* replace(tName.begin(), tName.end(), '\_', ' ');
* cout << "Occupied by " << tName << endl;
* if(area\_flag == true)
* {
* cout << "Error: Area should not be less than 50 sq. ft.\n";
* set\_area\_flag(false);
* }
* cout << "Area: " << area << " sq. ft." << endl;
* if(rent\_flag == true)
* {
* cout << "Error: Rent should not be less than $500.\n";
* set\_rent\_flag(false);
* }
* cout << "Rent: $" << rent << " ($" << getRentPerSqFoot(rent, area) << " per sq. ft.)" << endl << endl;
* }
* }
* #include <iostream>
* #include <cstdlib>
* #include <cctype>
* using namespace std;
* class Money
* {
* public:
* friend Money operator +(const Money& amount1, const Money& amount2);
* friend bool operator ==(const Money& amount1, const Money& amount2);
* Money(long dollars, int cents);
* Money(long dollars);
* Money();
* double get\_value() const;
* void input(istream$ ins);
* void output(ostream$ outs) const;
* private:
* long all\_cents;
* };
* int digit\_to\_int(char c);
* int main()
* {
* Money cost(1, 50), tax(0, 15), total;
* total = cost + tax;
* cout << "cost = ";
* cost.output(cout);
* cout << endl;
* cout << "tax = ";
* tax.output(cout);
* cout << endl;
* cout << "total bill = ";
* total.output(cout);
* cout << endl;
* if (cost == tax)
* cout << "Move to another state.\n";
* else
* cout << "Things seem normal.\n";
* return 0;
* }
* Money operator +(const Money& amount1, const Money& amount2)
* {
* Money temp;
* temp.all\_cents = amount1.all\_cents + amount2.all\_cents;
* return temp;
* }
* bool operator ==(const Money& amount1, const Money& amount2)
* {
* return (amount1.all\_cents == amount2.all\_cents);
* }
* Money::Money(long dollars, int cents)
* {
* if (dollars \* cents < 0) //If one is negative and one is positive
* {
* cout << "Illegal values for dollars and cents.\n";
* exit(1);
* }
* all\_cents = dollars \* 100 + cents;
* }
* Money::Money(long dollars) : all\_cents(dollars \* 100)
* {
* //Body intentionally blank.
* }
* Money::Money() : all\_cents(0)
* {
* //BOdy intentionally blank.
* }
* void Money::input(istream$ ins)
* {
* char one\_char, decimal\_point, digit1, digit2;
* long dollars;
* int cents;
* bool negative;
* ins >> one\_char;
* if (one\_char == ' ')
* {
* negative = true;
* ins >> one\_char;
* }
* else
* negative = false;
* ins >> dollars >> decimal\_point >> digit1 >> digit2;
* if (one\_char != '$' || decimal\_point != '.' || !isdigit(digit1) || !isdigit(digit2))
* {
* cout << "Error illegal form for money input\n";
* exit(1);
* }
* cents = digits\_to\_int(digit1) \* 10 + digit\_to\_int(digit2);
* all\_cents = dollars \* 100 + cents;
* if (negative)
* all\_cents = -all\_cents;
* }
* void Money::output(ostream& outs)
* {
* long positive\_cents, dollars, cents;
* positive\_cents = labs(all\_cents);
* dollars = positive\_cents / 100;
* cents = positive\_cents % 100;
* if (all\_cents < 0)
* outs << "-$" << dollars << '.';
* else
* outs << "$" << dollars << '.';
* if (cents < 10)
* outs << '0';
* outs << cents;
* }
* int digit\_to\_int(char c)
* {
* return (static\_cast<int>(c) - static\_cast<int>('0'));
* }