C++ Lecture 12

* Structures and Classes
* CIS 251 • Shelby-Hoover Campus

Grouping Variables

* In chapter 2, you learned how to declare individual variables of various types
* In chapter 7, you learned how to declare a set of variables of the same type with a single name
* In reality, programs need to store data of varying types on real-world objects (clients, employees, companies, orders)
* The ideal situation is to package this data into a single entity in which a program can hold all of the data related to one real-world object

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**

Defining a Structure

* Structure definitions are usually declared globally (before / outside functions) so that they will be available to every function
* The definition begins with the key word struct and the structure tag (usually beginning with an uppercase letter)
* In curly brackets, declare each member variable
* After the closing curly bracket, add a semicolon
* Form:  
    
  struct TypeName  
  {  
   memberType1 memberName1;  
   memberType2 memberName2;  
   …  
   memberTypeN memberNameN;  
  };

Structure Definition Example

* A structure type for certificates of deposit:  
    
  struct CDAccount  
  {  
   double balance;  
   double interest\_rate;  
   int term; // months until maturity  
  };

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
* A variable of one structure type can be used as a member variable of another structure type:  
    
  struct Name  
  {  
   string first;  
   char middle;  
   string last;  
  };  
    
  struct Student  
  {  
   Name studentName;  
   int age;  
   string major;  
  };

Structure Variables

* The structure tag serves as the type for the declaration of structure variables:  
    
  CDAccount sixMonth, oneYear;  
  Student aStudent;
* Each structure variable contains its own set of values for the member variables
* Structure variables may also be declared after the closing curly bracket and before the semicolon of the structure definition itself (the type is automatic)

Using Structure Variables

* Each member variable can be accessed using the dot operator, and each may be used according to its type:  
    
  sixMonth.balance = 500.00;  
  sixMonth.interest\_rate = 1.85;  
  sixMonth.term = 6;  
    
  aStudent.studentName.first = "John";  
  aStudent.studentName.middle = 'R';  
  aStudent.studentName.last = "Smith";  
  aStudent.age = 20;  
  aStudent.major = "Computer Science";

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 };
* An assignment statement involving two structure variables copies the values from the member variables of the structure variable on the right to the member variables of the structure variable on the left:  
    
  newAccount = oldAccount;

Structures and Functions

* A function can have a structure variable as a parameter (pass by value or by reference):  
    
  void getData(CDAccount& theAccount)  
  {  
   cout << "Enter account balance: $";  
   cin >> theAccount.balance;  
   cout << "Enter account interest rate: ";  
   cin >> theAccount.interest\_rate;  
   cout << "Enter the number of months until maturity" << endl  
   << "(must be 12 or fewer months): ";  
   cin >> theAccount.term;  
  }
* 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
* The textbook notes that C++ allows the use of the keyword struct with a class definition, but for simplicity assume that the syntax for each is separate

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**

Defining a Class

* The outer portion of a class definition mirrors that of a structure: the keyword class, the new type name (capitalized), an opening curly bracket, the class contents, a closing curly bracket, and a semicolon
* Members of a class are divided into public and private sections
  + Each section begins with public or private and a colon
  + public members can be accessed by external code using the dot operator
  + private members are only accessible to other members of the same class
  + If all of the members of a class are public variables (no member functions), it is the equivalent of a structure
  + Member variables are typically private; all access to these variables must be performed through public member functions that verify that the access to be performed is allowed

Member Function Definitions

* A class definition may include complete member function definitions or just their prototypes
* Separate member function definitions are usually written after the main function
* A member function definition written outside of the class definition must include a **type qualifier** and the **scope resolution operator** in its header
  + The type qualifier is the class name, listed after the return type and before the member function name
  + The scope resolution operator is two colons (::) and is placed between the type qualifier and the member function name
  + Form: returnType ClassName::functionName()

Class / Member Function Example

* DayOfYear with all public members:  
    
  #include <iostream>  
  using namespace std;  
    
  class DayOfYear  
  {  
  public:  
   void output(); // member function prototype  
   int month;  
   int day;  
  };  
    
  // main goes here  
    
  void DayOfYear::output() // member function definition  
  {  
   cout << "month = " << month << ", day = " << day << endl;  
  }

Declaring, Using Objects

* The class name serves as the type name for any variables to be declared
* public members may be accessed with the dot operator
* 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**
* The assignment operator for objects works just the same as the assignment operator for structure variables

Object Examples

* In this example, examDate and tornadoWarning are both DayOfYear objects:  
    
  DayOfYear examDate, tornadoWarning; // Declaring objects  
    
  examDate.month = 4; // Using public member variables  
  examDate.day = 11;  
    
  tornadoWarning = examDate; // Copying an object's variables  
    
  cout << "The exam date is as follows:\n";  
  examDate.output(); // Calling a public member function  
  cout << "The tornado warning was issued:\n";  
  tornadoWarning.output();

public vs. private

* Member variables are usually private
* Most member functions are public
* 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

* When member variables are private, a class usually contains public member functions that allow the user to obtain or modify their values
* An **accessor** member function (name begins with “get”) allows the user to access a value from within the object
  + Typically no parameters, with a return type to match the type of value to be accessed
  + The body is usually a simple return statement with either a member variable or a calculated value based on one or more member variables
* A **mutator** member function (name begins with “set”) allows the user to modify one or more values in an object
  + At least one parameter with a type to match the member variable being set, with void as the typical return type
  + The body may assign the parameter value(s) directly to the member variable(s), or there may be logic to check or modify the parameter value(s)

public / private Example

* An expanded DayOfYear class with private member variables:  
    
  class DayOfYear  
  {  
  public:  
   void setMonth(int newMonth);  
   void setDay(int newDay);  
   int getMonth();  
   int getDay();  
   void input();  
   void output();  
  private:  
   int month;  
   int day;  
  };

Definition Examples (1)

* Mutator member function definitions for DayOfYear:  
    
  void DayOfYear::setMonth(int newMonth)  
  {  
   if ((newMonth >= 1) && (newMonth <= 12))  
   month = newMonth;  
   else  
   cout << "Error! The month must be between 1 and 12." << endl;  
  }

void DayOfYear::setDay(int newDay)  
{  
 if ((newDay >= 1) && (newDay <= 31))  
 day = newDay;  
 else  
 cout << "Error! The day must be between 1 and 31." << endl;  
}

Definition Examples (2)

* Accessor member function definitions for DayOfYear:  
    
  int DayOfYear::getMonth()  
  {  
   return month;  
  }  
    
  int DayOfYear::getDay()  
  {  
   return day;  
  }

Definition Examples (3)

* Assume that the definition of output is the same as in the first version of DayOfYear
* The input member function definition for DayOfYear utilizes the mutator member functions to reject invalid input:  
    
  void DayOfYear::input()  
  {  
   int tempMonth, tempDay;  
    
   cout << "Enter the month as a number: ";  
   cin >> tempMonth;  
   setMonth(tempMonth);  
    
   cout << "Enter the day of the month: ";  
   cin >> tempDay;  
   setDay(tempDay);  
  }

Mutator Usage Example

* Now that month and day are private member variables, the following will not work:  
    
  examDate.month = 4;  
  examDate.day = 11;
* Instead, a program must use the public mutators:  
    
  examDate.setMonth(4);  
  examDate.setDay(11);
* Because the mutator member functions validate the values received by the parameters, the following erroneous arguments will be rejected:  
    
  examDate.setMonth(15);  
  examDate.setDay(-3);