Chapter 10

Defining Structures and Classes

Prof. Chien-Nan (Jimmy) Liu
Dept. of Electrical Engineering
National Chiao-Tung Univ.

Tel: (03)5712121 ext:31211
E-mail: jimmyliu@nctu.edu.tw
http://mseda.ee.nctu.edu.tw/jimmyliu

Overview

- *10.1 Structures*
- 10.2 Classes
- 10.3 Abstract Data Types
- 10.4 Introduction to Inheritance

Aggregate Data Type

- Array is a kind of "set" for many data
  - All elements are required to be the same type
- What if the data set has many different types??
  - Need an aggregate data type that supports mixed-type variables

Structure and Class

- A *class* is a data type whose variables are objects
  - Any pre-defined data type can be an object
    - Ex: int, char, double …
  - User-provided function can be an object
    - Easier to use this class
  - Those objects are called the "members" of this class
- A *structure* is a special class that contains no member functions
  - Contains multiple values of possibly different types
  - In legacy C, it supports structures only
- You can use pre-defined classes (ex: ifstream) or define your own classes

Class/Structure Definitions

- A class definition includes
  - A description of the kinds of values the variable can hold
  - A description of the member functions
- This definition does not reserve any space in memory → creates a new data type only
- Class variables should be declared like variables of other types → two-step declarations

Example of Structure Definition

- A bank Certificate of Deposit (CD) account often includes the following values:
  a balance
  an interest rate
  a term (months to maturity)
- The structure for a CD account can be defined as

```
struct CDAccount
{
        double balance;
        double interestRate;
        int term;  //months to maturity
};  ←   Remember this semicolon!
```

- Keyword *struct* begins a structure definition
- CDAccount will be used as the type name

Create Structure Objects

- Structure definition is generally placed outside any function definition
  - This makes the structure type available to all code that follows the structure definition
- In order to use the structure, create some objects based on the declaration first !!
- To declare two variables of type CDAccount:

```
CDAccount  myAccount, yourAccount;
```

  - *myAccount* and *yourAccount* contain their own member variables: *balance*, *interestRate*, and *term*
  - Now you can use those member variables …

Initializing Classes

- A structure can be initialized when declared
- Example:

```
struct Date
{
        int month;
        int day;
        int year;
};
```

  - Can be initialized in this way

```
Date   dueDate  =  { 12, 31, 2004 };
```

  - You can also assign the value for each member one by one

Using Member Variables

- Member variables are specific to the structure variable in which they are declared
  - Using dot (.) operator to specify a member variable:
    Structure_Variable_Name . Member_Variable_Name
- Ex: given the declaration:
  CDAccount myAccount, yourAccount;
  - Use the dot operator to specify a member variable, for example, myAccount.balance
- Member variables can be used just as any other variable of the same type
  - yourAccount.balance = 1000;
    myAccount.balance = yourAccount.balance + 2500;
  - myAccount.balance and yourAccount.balance are different variables!

Demonstrate CDAccount Structure

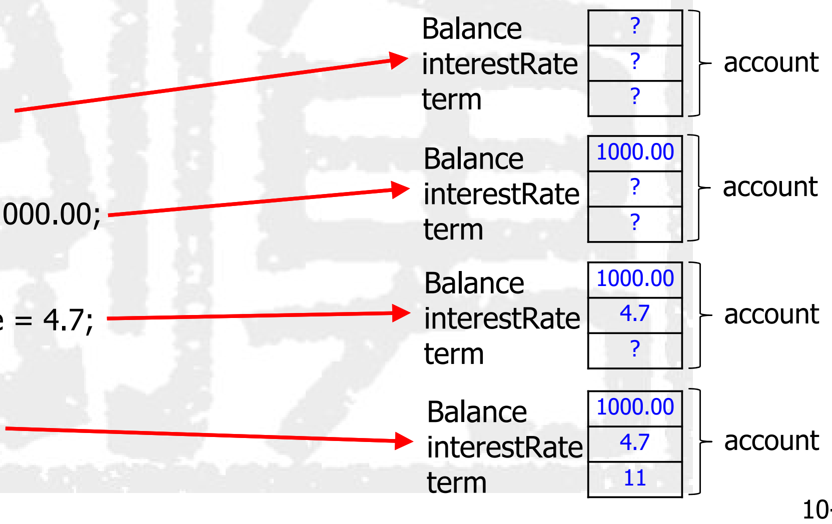
```
#include <iostream>
using namespace std;

struct CDAccount
{
    double balance;
    double interestRate;
    int term;   //months until maturity
};

void getData(CDAccount& theAccount);
int main( )
{
    CDAccount account;
    getData(account);

    double rate, interest;
    rate = account.interestRate/100.0;
    interest = account.balance*rate*
                (account.term/12.0);
    account.balance = account.balance +
                         interest;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "When your CD matures in "
        << account.term << " months,\n"
        << "it will have a balance of $"
        << account.balance << endl;
    return 0;
}

void getData(CDAccount& theAccount)
{
    cout << "Enter account balance: $";
    cin >> theAccount.balance;

    cout << "Enter account interest rate: ";
    cin >> theAccount.interestRate;
    cout << "Enter the number of months until”
          << "maturity\n"
          << "(must be 12 or fewer months): ";
    cin >> theAccount.term;
}
```

Program Results & Member Values

```
struct CDAccount
{
    double balance;
    double interestRate;
    int term; //months until maturity
};

int main()
{
    CDAccount account;
        . . .
    account.balance = 1000.00;

    account.interestRate = 4.7;

    account.term = 11;
```

Sample Dialogue

```
Enter account balance: $100.00
Enter account interest rate: 10.0
Enter the number of months until maturity
(must be 12 or fewer months): 6
When your CD matures in 6 months,
it will have a balance of $105.00
```



Duplicate Names

- Member variable names duplicated between structure types are not a problem

```
struct FertilizerStock
{
    double quantity;
    double nitrogenContent;
};

FertilizerStock  superGrow;
```

```
struct CropYield
{
   int quantity;
   double size;
};

CropYield  apples;
```

- superGrow.quantity and apples.quantity are different variables stored in different locations
  - Your ice cream ≠ my ice cream

Structures as Arguments

- Structures can be arguments in function calls
  - Just pass it as a typical variable
- By default, entire structures are passed by value
  - Called-by-reference is recommended because a structure variable may occupy large memory space
- Example:

```
void getData(CDAccount& theAccount);
```

  - Uses the structure type CDAccount we saw earlier as the type for a call-by-reference parameter

Structures as Return Types

- Structures can be the type of a value returned by a function
- Example:

```
CDAccount Wrap(double Balance, double Rate, int Term)
{
    CDAccount temp;
    temp.balance = Balance;
    temp.interestRate = Rate;
    temp.term = Term;
    return temp;
}
```

- Use Wrap to give the value of a CDAccount variable:

```
CDAccount  newAccount;
newAccount = Wrap(1000.00, 5.1, 11);
```

Assignment and Structures

- The assignment operator (=) can be used to assign values to structure types
- You can assign a value to each member individually

```
CDAccount myAccount, yourAccount;
myAccount.balance = 1000.00;
myAccount.interestRate = 5.1;
myAccount.term = 12;
```

- You can also assign the whole structure to another structure variable of the same type
  - Ex: yourAccount = myAccount;

    Member-by-member copy values from myAccount to yourAccount

Hierarchical Structures

- Structures can contain member variables that are also structures

```
struct Date
{
    int month;
    int day;
    int year;
};
```

```
struct PersonInfo
{
    double height;
    int weight;
    Date birthday;
};
```

  - struct PersonInfo contains a Date structure
- Given a variable of type *PersonInfo:*

```
PersonInfo person1;
```

  - The birthday member is still a structure
- To know the exact number of year, we specify the year member of the birthday member

```
cout << person1.birthday.year;   // two-level dots
```

Overview

- 10.1 Structures
- *10.2 Classes*
- 10.3 Abstract Data Types
- 10.4 Introduction to Inheritance

Classes

- You only need to know what the program does, not how it does it
  - Do you know how to build a car? You don't have to …
  - Simply use the user-friendly "interfaces" to the car's complex internal mechanisms
- What must happen before you can do this?
  - Good user interface → member functions
  - Good package for easy use → information hiding
- This is called "encapsulation" in C++ class
  - Combining a number of items, such as variables and functions, into a single package
  - Class ≈ a structure definition plus member functions

Defining a Class

- You cannot drive the engineering drawings of a car
  - The car must be built from the engineering drawings before you drive it
  - Just like structure, 2-step declaration is required for class
- Ex: create a class definition named *DayOfYear*
  - Decide on the values to represent
    - An integer for the number of the month, an integer for the number of days → data member
  - Decide on the member functions
    - Just one member function named output for printing results

```
class DayOfYear
{
    public:
        void output( );
        int month;
        int day;
};
```

member function declaration

Defining a Member Function

- Member functions are declared in the class declaration
  - Identify the class in which the function is a member
  - You can use all data members without extra declaration
- Member function definition syntax:

```
Returned_Type Class_Name::Function_Name(Parameter_List)
{
        Function Body Statements
}
```

  - Ex:

```
void DayOfYear::output( )
{
    cout << "month = " << month
            << ", day = " << day << endl;
}
```

The '::' Operator

- '::' is the scope resolution operator
  - Tells the class a member function is a member of
  - void DayOfYear::output( ) indicates that function *output* is a member of the *DayOfYear* class
- Comparison of '::' and '.'
  - '::' used with classes to identify a member

```
void DayOfYear::output( )
{
        // function body
}
```

  - '.' used with variables to identify a member

```
DayOfYear birthday;
birthday.output( );
```

Declaring a Class Object

- Once a class is defined, an object of the class is declared just as variables of any other type

```
class Bicycle
{
    // class definition lines
};
Bicycle myBike, yourBike;
```

- Objects and structures can be assigned values with the assignment operator (=) → only data is copied

```
DayOfYear  dueDate, tomorrow;
tomorrow.set(11, 19);
dueDate = tomorrow;
```

Calling Member Functions

- In a class object, using its data members can be done in this way:

  DayOfYear today, birthday;
  cout << “Today is ” << today.day;

- Calling the member function of a class can be done in this way:

  today.output( );
  birthday.output( );

  Called in the same way as typical functions, except the dot (.) before function name

  - Note that today and birthday are two different objects
  - They have their own versions of the month and day variables for use by the output function

Code: Class with Member Functions

```
#include <iostream>
using namespace std;
class DayOfYear
{
public:
    void output( );
    int month;
    int day;
};

int main( )
{
    DayOfYear today, birthday;

    cout << "Enter today's date:\n";
    cout << "Enter month as a number: ";
    cin >> today.month;
    cout << "Enter the day of the month: ";
    cin >> today.day;
    cout << "Enter your birthday:\n";
    cout << "Enter month as a number: ";
    cin >> birthday.month;
    cout << "Enter the day of the month: ";
    cin >> birthday.day;
    cout << "Today's date is ";
    today.output( );
    cout << "Your birthday is ";
    birthday.output( );

    if (today.month == birthday.month
        && today.day == birthday.day)
        cout << "Happy Birthday!\n";
    else
        cout << "Happy Unbirthday!\n";

    return 0;
}

//Uses iostream:
void DayOfYear::output( )
{
    cout << "month = " << month
         << ", day = " << day << endl;
}
```

Sample Dialogue

```
Enter today's date:
Enter month as a number: 10
Enter the day of the month: 15
Enter your birthday:
Enter month as a number: 2
Enter the day of the month: 21
Today's date is month = 10, day = 15
Your birthday is month = 2, day = 21
Happy Unbirthday!
```

Problems with DayOfYear

- Member functions can be used as black boxes → information hiding
  - The function can be used without knowing how it is coded
  - Allow programmers to easily change or improve a class without forcing users changing what they have done
- However, in previous example, the data stored in the object (ex: month) can be changed by anyone
  - Users still have to know how the data is stored legally
- It is better to add member functions for accessing or changing the member variables
  - Directly reference to the member variables is not allowed!
  - More safe to protect the data from illegal changes

Data Protection in C++ Class

- C++ helps us restrict the program from directly referencing member variables
- Keyword *private* identifies the members that can be accessed only by member functions of the same class
  - Members that follow the keyword private are private members of the class
- Keyword *public* identifies the members that can be accessed from outside the class
  - Members that follow the keyword public are public members of the class

General Class Definitions

- The syntax for a class definition is

```
class Class_Name
{
  public:
          Member_Specification_1
          Member_Specification_2
          …
          Member_Specification_3
  private:
          Member_Specification_n+1
          Member_Specification_n+2
          …
};
```

Using Private Variables

- Changing the values of private variables requires the use of public member functions of the class

```
void DayOfYear::set(int new_month, int new_day)
{
    month = new_month;
    day = new_day;
}
```

- The new DayOfYear class is demonstrated
  - Uses all private member variables for protection
  - Uses member functions to do all manipulation of the private member variables
    - Interface (member functions) and internal data (member variables) are implemented separately

Demonstrate the Class DayOfYear (1/2)

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

class DayOfYear
{
public:
    void input( );
    void output( );
    void set(int newMonth, int newDay);
    int getMonth( );
    int getDay( );
private:
    void checkDate( );
    int month;
    int day;
};
```

```
int main( )
{
    DayOfYear today, bachBirthday;
    cout << "Enter today's date:\n";
    today.input( );
    cout << "Today's date is ";
    today.output( );

    bachBirthday.set(3, 21);
    cout << "J. S. Bach's birthday is ";
    bachBirthday.output( );

    if ( today.getMonth( ) == bachBirthday.getMonth( ) &&
             today.getDay( ) == bachBirthday.getDay( ) )
        cout << "Happy Birthday Johann Sebastian!\n";
    else
        cout << "Happy Unbirthday Johann Sebastian!\n";
    return 0;
}
```

Demonstrate the Class DayOfYear (2/2)

```
void DayOfYear::input( )
{
    cout << "Enter the month as a number: ";
    cin >> month;
    cout << "Enter the day of the month: ";
    cin >> day;
    checkDate( );
}

void DayOfYear::set(int newMonth, int newDay)
{
    month = newMonth;
    day = newDay;
    checkDate();
}

int DayOfYear::getMonth( )
{
    return month;
}
```

```
int DayOfYear::getDay( )
{
    return day;
}

void DayOfYear::output( )   // same as before

void DayOfYear::checkDate( )
{
    if ((month < 1) || (month > 12) ||
        (day < 1) || (day > 31))
    {
        cout << "Illegal date. Aborting program.\n";
        exit(1);
    }
}
```

Sample Dialogue

```
Enter today's date:
Enter the month as a number: 3
Enter the day of the month: 21
Today's date is month = 3, day = 21
J. S. Bach's birthday is month = 3, day = 21
Happy Birthday Johann Sebastian!
```

Private or Public ??

- In most cases, member variables are set as private to protect them from illegal changes
  - However, public data member is still allowed as in structure
- Member functions are often set as public to change or obtain the values of private variables
  - Which functions are required?? → try set and get
  - Set functions allow you to change the values of member variables, ex: *set(…)* in class DayOfYear
  - Get functions allow you to obtain the values of member variables, ex: *getDay()* in class DayOfYear
- Private member functions are also allowed
  - Called utility function or helper function for internal use

Using Private Member Functions

- While calling a public member function from the main program, you must include the object name:

  Ex: account1.update( );
- When a member function calls a private member function in the same class, object name is not used
  - Given a private function of the *BankAccount* class

    fraction (double percent);

    fraction can be called by other member function as:

```
void BankAccount::update( )
{
    balance = balance +
            fraction(interestRate)* balance;
}
```

Example: BankAccount Class

- This bank account class allows
  - Withdrawal of money at any time
  - All operations normally expected of a bank account
    - Implemented with member functions
  - Storing an account balance
  - Storing the account's interest rate

Sample dialogue

```
Start of Test:
account1 initial statement:
Account balance $123.99
Interest rate 3.00%
account1 with new setup:
Account balance $100.00
Interest rate 5.00%
account1 after update:
Account balance $105.00
Interest rate 5.00%
account2:
Account balance $105.00
Interest rate 5.00%
```

Demo BankAccount Class (1/2)

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

class BankAccount
{
public:
    void set(int usd, int cent, double rate);
    void set(int usd, double rate);
    // Two versions for seting interest rate
    void update( );   // add one year interest
    double getBalance( );  // current amount
    double getRate( );   // current rate
    void output(ostream& outs);
private:
    double balance;
    double interestRate;
    double fraction(double percent);
    //Converts a percentage to a fraction
};
```

```
void BankAccount::set(int usd, int cent, double rate)
{
    if ((usd < 0) || (cent < 0) || (rate < 0))  {
        cout << "Illegal values for money or"
             << " interest rate.\n";
        return;
    }
    balance = usd + 0.01*cent;
    interestRate = rate;
}

void BankAccount::set(int usd, double rate)
{
    if ((usd < 0) || (rate < 0))  {
        cout << "Illegal values for money or"
             << " interest rate.\n";
        return;
    }
    balance = usd;
    interestRate = rate;
}
```

Demo BankAccount Class (2/2)

```
void BankAccount::update( )
{    balance = balance +
               fraction(interestRate)*balance;  }

double BankAccount::fraction(double percent)
{    return (percent/100.0);    }

double BankAccount::getBalance( )
{    return balance;    }

double BankAccount::getRate( )
{    return interestRate;    }

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

```
int main( )
{
    BankAccount account1, account2;
    cout << "Start of Test:\n";
    account1.set(123, 99, 3.0);
    cout << "account1 initial statement:\n";
    account1.output(cout);

    account1.set(100, 5.0);
    cout << "account1 with new setup:\n";
    account1.output(cout);

    account1.update( );
    cout << "account1 after update:\n";
    account1.output(cout);

    account2 = account1;
    cout << "account2:\n";
    account2.output(cout);
    return 0;
}
```

Constructors

- A constructor can be used to initialize member variables when an object is declared
  - A constructor is a member function that is often public
  - A constructor is automatically called when an object of the class is declared
  - A constructor's name must be the name of the class
  - A constructor cannot return a value
    - No return type, not even void, in defining a constructor

```
Ex: class BankAccount
    {
        public:
            BankAccount(int usd, int cent, double rate);
            …
    };
```

Constructor Definition

- The constructor for the BankAccount class could be defined as

```
BankAccount::BankAccount(int usd, int cent, double rate)
{
    if ((usd < 0) || (cent < 0) || ( rate < 0 ))
    {
        cout << "Illegal values for money or rate\n";
         exit(1);
    }
    balance = usd + 0.01 * cent;
    interestRate = rate;
}
```

  - Note that the class name and function name are the same

Calling A Constructor

- A constructor is not called like a normal member function:

```
BankAccount  account1;
account1.BankAccount(10, 50, 2.0);
```
✗

- A constructor is automatically called in the object declaration

```
BankAccount account1(10, 50, 2.0);
```
○

  - Creates a BankAccount object and calls the constructor to initialize the member variables

Overloading Constructors

- Constructors can be overloaded by defining constructors with different parameter lists
  - Possible constructors for BankAccount might be

```
BankAccount (double balance, double interestRate);
BankAccount (double balance);
BankAccount ( );   // default constructor
```

- The default constructor is called when no argument is given during object declaration (better to have)

```
BankAccount::BankAccount( )
{
        balance = 0;
        rate = 0.0;
}
```

Demo Class with Constructors (1/2)

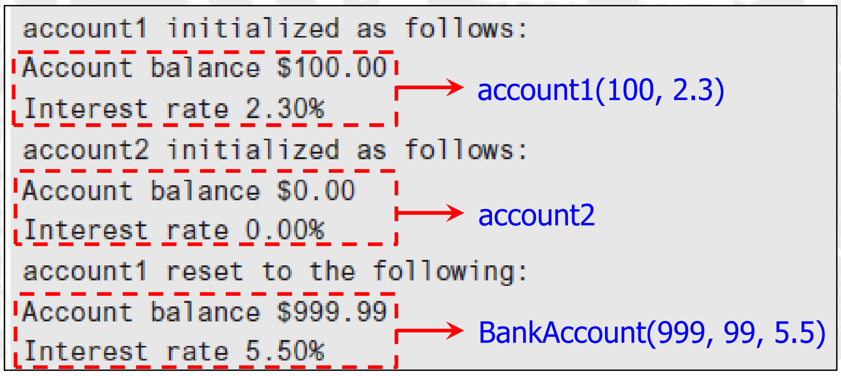
```
#include <iostream>
using namespace std;

class BankAccount
{
public:
   BankAccount(int usd, int cent, double rate);
   BankAccount(int usd, double rate);
   BankAccount( );
   void set(int usd, int cent, double rate);
   void set(int usd, double rate);   // set rate
   void update( );   // add one year interest
   double getBalance( );  // current amount
   double getRate( );   // current rate
   void output(ostream& outs);
private:
   double balance;
   double interestRate;
   double fraction(double percent);
   //Converts a percentage to a fraction
};
```

```
BankAccount::BankAccount(int usd, int cent,
                                    double rate)
{
   if ((usd < 0) || (cent < 0) || (rate < 0)) {
      cout << "Illegal values for money or "
           << "interest rate.\n";
      return;
   }
   balance = usd + 0.01*cent;
   interestRate = rate;
}

BankAccount::BankAccount(int usd, double rate)
{
   if ((usd < 0) || (rate < 0)) {
      cout << "Illegal values for money or "
           << "interest rate.\n";
      return;
   }
   balance = usd;  interestRate = rate;
}
```

Demo Class with Constructors (2/2)

```
BankAccount::BankAccount( ) : balance(0),
                              interestRate(0.0)
{
   //Body intentionally empty
}

int main( )
{
   BankAccount account1(100, 2.3), account2;
   cout << "account1 initialized as follows:\n";
   account1.output(cout);
   cout << "account2 initialized as follows:\n";
   account2.output(cout);

   account1 = BankAccount(999, 99, 5.5);
   cout << "account1 reset to the following:\n";
   account1.output(cout);
   return 0;
}
```

default constructor



```
account1 initialized as follows:
Account balance $100.00
Interest rate 2.30%
account2 initialized as follows:
Account balance $0.00
Interest rate 0.00%
account1 reset to the following:
Account balance $999.99
Interest rate 5.50%
```

Initialization Sections

- An initialization section in a function provides an alternative way to set default values of the members

  BankAccount::BankAccount( ): balance(0), interestRate(0.0);
  { // No code needed in this example }

  - Values in parenthesis are the initial values for the variables

- Member functions with parameters can also use initialization sections

  BankAccount::BankAccount(int usd, int cent, double rate)
  : balance (usd + 0.01 * cent), interestRate(rate)
  { // code is the same as in previous example }

  - The parameters can be arguments in the initialization

Destructors

- Opposite to constructor, the destructor is called implicitly when an object is destroyed
  - For example, an automatic object is destroyed when program execution leaves its scope (ex: function)
  - Often used to delete the dynamic variables automatically
- The name of the destructor for a class is the tilde character (~) followed by the class name
  - Ex: BankAccount::~BankAccount()
- A destructor receives no parameter and returns no value → cannot specify a return type, not even void
- If you do not explicitly provide a destructor, the compiler creates an “empty” destructor → do nothing

Access Members from Pointer

- The dot operator (.) after an object’s name is able to access the object’s members
- The arrow operator (->) preceded by a pointer to an object is able to access the object’s members
  - Ex:
```
BankAccount a, *aptr = &a;
a.set(100, 2.3);
double rate = aptr->getRate( );   // (*aptr).getRate( )
```
- Arrow operator is useful when only pointer is passed to another functions (ex: dynamic variables)
- The following example compares the usage of accessing class members by reference and pointers

Demo Member Access Methods (1/2)

```
// Demonstrating the class member access operators . and ->
#include <iostream>
using namespace std;

//class Count definition
class Count
{
public:  // public data is dangerous
    // sets the value of private data member x
    void setX( int value )
    {
        x = value;
    }  // end function setX

    //prints the value of private data member x
    void print()
    {
        cout << x << endl;
    }  // end function print
```

Demo Member Access Methods (2/2)

```
private:
    int x;
};  // end class Count

int main()
{
    Count counter; // create counter object
    Count *counterPtr = &counter;  // create pointer to counter
    Count &counterRef = counter;  // create reference to counter

    cout << "Set x to 1 and print using the object's name: ";
    counter.setX( 1 );  // set data member x to 1
    counter.print();  // call member function print

    cout << "Set x to 2 and print using a reference to an object: ";
    counterRef.setX( 2 );  // set data member x to 2
    counterRef.print();  // call member function print

    cout << "Set x to 3 and print using a pointer to an object: ";
    counterPtr -> setX( 3 );  // set data member x to 3
    counterPtr -> print();  // call member function print
}  // end main
```

Sample Dialogue

```
Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3
```