Chapter 10

Defining Classes

What Is a Class?

- A class is a data type whose variables are objects
- Some pre-defined data types you have used are
  - `int`
  - `char`
- You can define your own classes
  - define your own types
  - compare with pre-defined data types, define new names for existing types, etc.

Class Definitions

- A class definition includes
  - A description of the kinds of values the variable can hold
  - A description of the member functions
- We will start by defining structures as a first step toward defining classes

Overview

10.1 Structures

10.2 Classes

10.3 Abstract Data Types

10.4 Introduction to Inheritance

10.1

Structures

Structures

- A structure can be viewed as an object
  - Contains no member functions
    (The structures used here have no member functions)

  - Contains multiple values of possibly different types
    - The multiple values are logically related as a single item
    - Example: A bank Certificate of Deposit (CD)
      has the following values:
      a balance
      an interest rate
      a term (months to maturity)

The CD Definition

- The Certificate of Deposit structure can be defined as

```
struct CDAccount
{
        double balance;
        double interest_rate;
        int term;  //months to maturity
};
```

Remember this semicolon!

- Keyword `struct` begins a structure definition
- `CDAccount` is the structure tag
- Member names are identifiers declared in the braces

Using the Structure

- **Structure definition is generally placed outside any function definition**
  - This makes the structure type available to all code that follows the structure definition
- To declare two variables of type CDAccount:

```
CDAccount  my_account, your_account;
```

  - `My_account` and `your_account` contain distinct member variables `balance`, `interest_rate`, and `term`

The Structure Value

- The Structure Value
  - Consists of the values of the member variables of the structure

- The value of an object of type `CDAccount`
  - Consists of the values of the member variables

```
balance
interest_rate
term
```

Specifying Member Variables

- Member variables are specific to the structure variable in which they are declared
  - Syntax to specify a member variable:
    ```
    Structure_Variable_Name.Member_Variable_Name
    ```
  - Given the declaration:
    ```
    CDAccount  my_account, your_account;
    ```
    - Use the dot operator to specify a member variable
      ```
      my_account.balance
      my_account.interest_rate
      my_account.term
      ```

Using Member Variables

- Member variables can be used just as any other variable of the same type
  - `my_account.balance = 1000;`
    `your_account.balance = 2500;`
    - Notice that `my_account.balance` and `your_account.balance` are different variables!
  - `my_account.balance = my_account.balance + interest;`

Display 10.1 (1)

Display 10.1 (2)

A Structure Definition (*part 1 of 2*)

```
//Program to demonstrate the CDAccount structure type.
#include <iostream>
using namespace std;

//Structure for a bank certificate of deposit:
struct CDAccount
{
    double balance;
    double interest_rate;
    int term;//months until maturity
};

void get_data(CDAccount& the_account);
//Postcondition: the_account.balance and the_account.interest_rate
//have been given values that the user entered at the keyboard.


int main( )
{
    CDAccount account;
    get_data(account);

    double rate_fraction, interest;
    rate_fraction = account.interest_rate/100.0;
    interest = account.balance*rate_fraction*(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;
}
```

Display 10.1
(1/2)

Display 10.1
(2/2)

A Structure Definition (*part 2 of 2*)

```
//Uses iostream:
void get_data(CDAccount& the_account)
{
    cout << "Enter account balance: $";
    cin >> the_account.balance;
    cout << "Enter account interest rate: ";
    cin >> the_account.interest_rate;
    cout << "Enter the number of months until maturity\n"
         << "(must be 12 or fewer months): ";
    cin >> the_account.term;
}
```

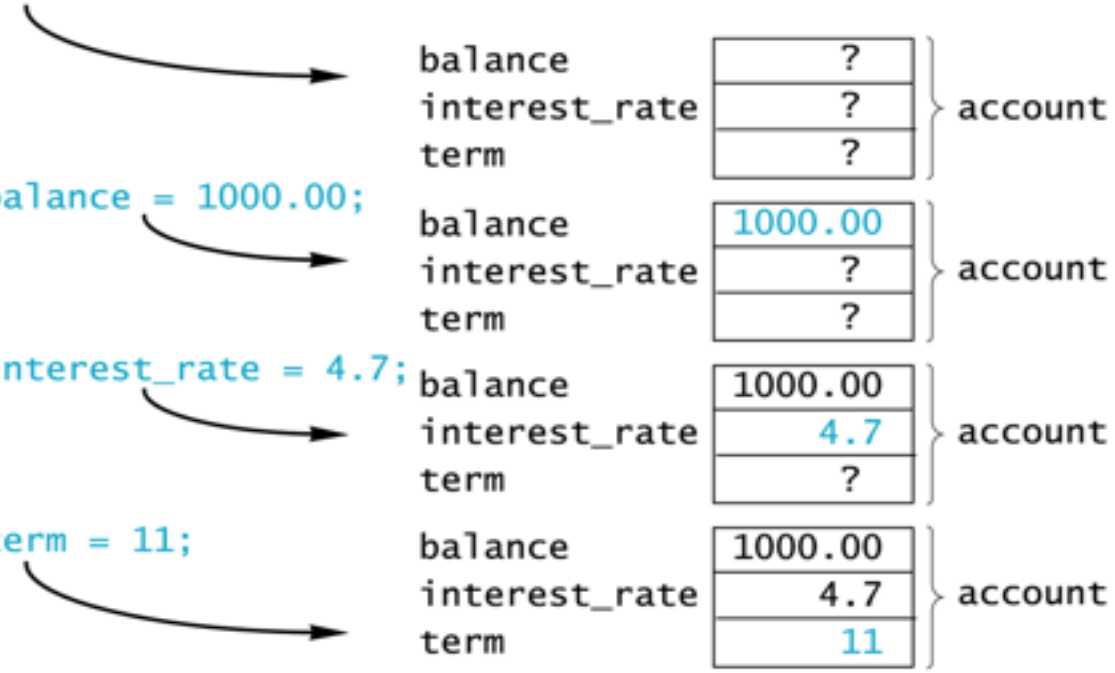
Sample Dialogue

```
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
```

Display 10.2

Member Values

```
struct CDAccount
{
    double balance;
    double interest_rate;
    int term;//months until maturity
};
int main()
{
   CDAccount account;
       ...
   account.balance = 1000.00;
   account.interest_rate = 4.7;
   account.term = 11;
```



Duplicate Names

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

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

FertilizerStock  super_grow;
```

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

CropYield  apples;
```

- `super_grow.quantity` and `apples.quantity` are different variables stored in different locations

Structures as Arguments

- Structures can be arguments in function calls
  - The formal parameter can be call-by-value
  - The formal parameter can be call-by-reference
- Example:

```
void get_data(CDAccount& the_account);
```

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

Assignment and Structures

- The assignment operator can be used to assign values to structure types
- Using the CDAccount structure again:

```
CDAccount my_account, your_account;
my_account.balance = 1000.00;
my_account.interest_rate = 5.1;
my_account.term = 12;
your_account = my_account;
```

  - Assigns all member variables in `your_account` the corresponding values in `my_account`

Structures as Return Types

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

```
CDAccount shrink_wrap(double the_balance,
                      double the_rate,
                      int the_term)
{
    CDAccount temp;
    temp.balance = the_balance;
    temp.interest_rate = the_rate;
    temp.term = the_term;
    return temp;
}
```

STOPPED HERE ON Feb 7.

Using Function shrink_wrap

- `shrink_wrap` builds a complete structure value in `temp`, which is returned by the function
- We can use `shrink_wrap` to give a variable of type `CDAccount` a value in this way:

```
CDAccount  new_account;
new_account = shrink_wrap(1000.00, 5.1,
  11);
```

The above assignment operator copies the whole structure content (given by the return statement) into new_account.

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

Using PersonInfo

- A variable of type `PersonInfo` is declared by

```
PersonInfo person1;
```

- To display the birth year of `person1`, first access the `birthday` member of `person1`

```
cout <<  person1.birthday…
```

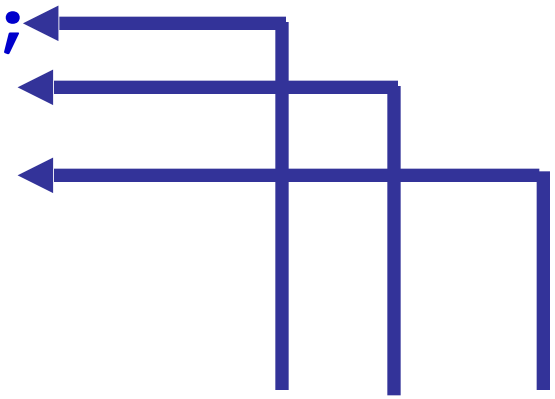
- But we want the year, so we now specify the year member of the birthday member

```
cout << person1.birthday.year;
```

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  due_date = {12, 31, 2004};
```

Compare with array initialization

Section 10.1 Exercise

- Can you
  - Write a definition for a structure type for records consisting of a person's wage rate, accrued vacation (in whole days), and status (hourly or salaried). Represent the status as one of the two character values 'H' and 'S'. Call the type EmployeeRecord.

10.2 Classes

Classes

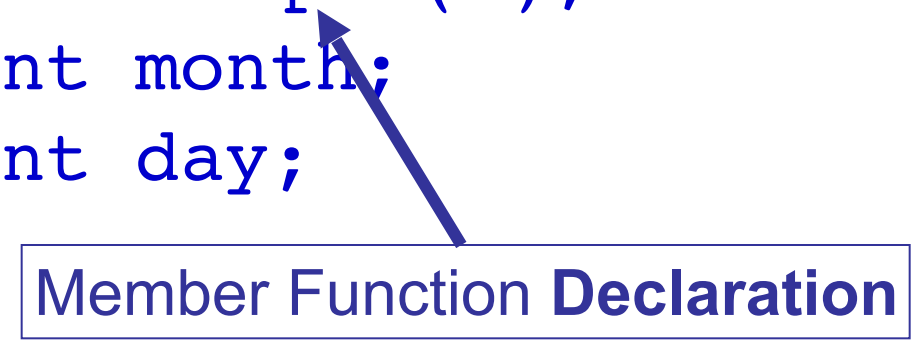
- A class is a data type whose variables are called objects
  - The definition of a class includes
    - Description of the kinds of values of the member variables
    - Description of the member functions
  - **A class description is somewhat like a structure definition plus the member functions**

A Class Example

- To create a new type named `DayOfYear` as a class definition
  - Decide on the values to represent
  - This example's values are dates such as July 4 using an integer for the number of the `month`
    - Member variable `month` is an int (Jan = 1, Feb = 2, etc.)
    - Member variable `day` is an int
  - Decide on the member functions needed
  - We use just one member function named `output`

Class DayOfYear Definition

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



Public or Private Members

- The keyword `public` identifies the members of a class that can be accessed from outside the class
  - Members that follow the keyword `public` are public members of the class
- The keyword `private` identifies the members of a class that can be accessed only by member functions of the class
  - Members that follow the keyword `private` are private members of the class

Defining a Member Function

- Member functions are **declared** in the class declaration
- Member function **definitions** identify the class in which the function is a member

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

Member Function Definition

- Member function definition syntax:

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

  - Example:

```
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
  - The class name that precedes '::' is a type qualifier

'::' and '.'

:: used with **classes** to identify a member

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

- used with **variables (or objects)** to identify a member

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

Calling Member Functions

- Calling the `DayOfYear` member function `output` is done in this way:

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

  - Note that `today` and `birthday` have their own versions of the `month` and `day` variables for use by the `output` funct

Display 10.3 (1)

Display 10.3 (2)

DISPLAY 10.3 Class with a Member Function *(part 1 of 2)*

```
1  //Program to demonstrate a very simple example of a class.
2  //A better version of the class DayOfYear will be given in Display 10.4.
3  #include <iostream>
4  using namespace std;
5  class DayOfYear
6  {
7  public:
8      void output( );
9      int month;
10     int day;
11 };
12 int main( )
13 {
14     DayOfYear today, birthday;
15     cout << "Enter today's date:\n";
16     cout << "Enter month as a number: ";
17     cin >> today.month;
18     cout << "Enter the day of the month: ";
19     cin >> today.day;
20     cout << "Enter your birthday:\n";
21     cout << "Enter month as a number: ";
22     cin >> birthday.month;
23     cout << "Enter the day of the month: ";
24     cin >> birthday.day;
25     cout << "Today's date is ";
26     today.output( );
27     cout << "Your birthday is ";
28     birthday.output( );
29     if (today.month == birthday.month
30         && today.day == birthday.day)
31         cout << "Happy Birthday!\n";
32     else
33         cout << "Happy Unbirthday!\n";
34     return 0;
35 }
36 //Uses iostream:
37 void DayOfYear::output( )
38 {
39     cout << "month = " << month
40          << ", day = " << day << endl;
41 }
```

Member function declaration

Calls to the member function **output**

Member function definition

(continued)

Display 10.3 (1/2)

Display 10.3
(2/2)

DISPLAY 10.3 Class with a Member Function *(part 2 of 2)*

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!
```

Encapsulation

- Encapsulation is
  - Combining a number of items, such as variables and functions, into a single package such as an object of a class

Problems With `DayOfYear`

- Changing how the `month` is stored in the class `DayOfYear` requires changes to the main program
- If we decide to store the `month` as three characters (JAN, FEB, etc.) instead of an `int`
  - `cin >> today.month` will no longer work because we now have three character variables to read
  - `if(today.month == birthday.month)` will no longer work to compare months
  - The member function "`output`" no longer works

Ideal Class Definitions

- Changing the implementation of `DayOfYear` requires changes to the program that uses `DayOfYear`
- An ideal class definition of `DayOfYear` could be changed without requiring changes to the program that uses `DayOfYear`

Fixing `DayOfYear`

- To fix `DayOfYear`
  - We need to add member functions to use when changing or accessing the member variables
    - If the program (that uses `DayOfYear`) **never directly references** the member variables of `DayOfYear`, changing how the variables are stored will not require changing the program
  - We need to be sure that the program does not ever directly reference the member variables

Public Or Private?

- C++ helps us restrict the program from **directly referencing** member variables
  - **Private** members of a class can only be referenced within the definitions of member functions
    - If the program (other than through member functions) tries to access a private member, the compiler gives an error message
  - **Private** members can be variables or functions

Private Variables

- Private variables cannot be accessed directly by the program
  - Changing their values requires the use of public member functions of the class
  - To set the private `month` and `day` variables in a new `DayOfYear` class use a member function such as

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

Public or Private Members

- The keyword `private` identifies the members of a class that can be accessed only by member functions of the class
  - Members that follow the keyword `private` are private members of the class
- The keyword `public` identifies the members of a class that can be accessed from outside the class
  - Members that follow the keyword `public` are public members of the class

A New `DayOfYear`

- The new `DayOfYear` class demonstrated in Display 10.4…
  - All member variables are private
  - Uses member functions to do all manipulation of the **private** member variables
    - **Private** member variables and member function definitions can be changed without changes to the program that uses `DayOfYear`

Display 10.4 (1)

Display 10.4 (2)

DISPLAY 10.4 Class with Private Members *(part 1 of 2)*

```
1  //Program to demonstrate the class DayOfYear.
2  #include <iostream>
3  using namespace std;
4  class DayOfYear
5  {
6  public:
7      void input( );
8      void output( );
9      void set(int new_month, int new_day);
10     //Precondition: new_month and new_day form a possible date.
11     //Postcondition: The date is reset according to the arguments.
12     int get_month( );
13     //Returns the month, 1 for January, 2 for February, etc.
14     int get_day( );
15     //Returns the day of the month.
16 private:
17     void check_date( );
18     int month;
19     int day;
20 };
21 int main( )
22 {
23     DayOfYear today, bach_birthday;
24     cout << "Enter today's date:\n";
25     today.input( );
26     cout << "Today's date is ";
27     today.output( );
28     bach_birthday.set(3, 21);
29     cout << "J. S. Bach's birthday is ";
30     bach_birthday.output( );
31     if ( today.get_month( ) == bach_birthday.get_month( ) &&
32               today.get_day( ) == bach_birthday.get_day( ) )
33         cout << "Happy Birthday Johann Sebastian!\n";
34     else
35         cout << "Happy Unbirthday Johann Sebastian!\n";
36     return 0;
37 }
38 //Uses iostream:
39 void DayOfYear::input( )
40 {
41     cout << "Enter the month as a number: ";
```

This is an improved version of the class `DayOfYear` *that we gave in Display 10.3.*

Private member function

Private member variables

(continued)

Display 10.4 (1/2)

DISPLAY 10.4 Class with Private Members *(part 2 of 2)*

```
42      cin >> month;
43      cout << "Enter the day of the month: ";
44      cin >> day;
45      check_date( );
46  }
47
48  void DayOfYear::output( )
     <The rest of the definition of DayOfYear::output is given in Display 10.3.>
49
50  void DayOfYear::set(int new_month, int new_day)
51  {
52      month = new_month;
53      day = new_day;
54      check_date();
55  }
56
57  void DayOfYear::check_date( )
58  {
59      if ((month < 1) || (month > 12) || (day < 1) || (day > 31))
60      {
61          cout << "Illegal date. Aborting program.\n";
62          exit(1);
63      }
64  }
65
66  int DayOfYear::get_month( )
67  {
68      return month;
69  }
70
71  int DayOfYear::get_day( )
72  {
73      return day;
74  }
```

Private members may be used in member function definitions (but not elsewhere).

A better definition of the member function `input` *would ask the user to reenter the date if the user enters an incorrect date.*

The member function `check_date` *does not check for all illegal dates, but it would be easy to make the check complete by making it longer. See Self-Test Exercise 14.*

The function `exit` *is discussed in Chapter 6. It ends the program.*

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!
```

Display 10.4 (2/2)

Using Private Variables

- **It is normal to make all member variables private**
- Private variables require member functions to perform all changing and retrieving of values
  - **Accessor** functions allow you to obtain the values of member variables
    - Example: `get_day` in class `DayOfYear`
  - **Mutator** functions allow you to change the values of member variables
    - Example: `set` in class `DayOfYear`

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
        …
 };
```

Declaring an Object

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

To create two objects of type Bicycle:

```
class Bicycle
{
      // class definition lines
};

Bicycle my_bike,  your_bike;
```

The Assignment Operator

- Objects and structures can be assigned values with the assignment operator (=)
  - Example:

```
DayOfYear  due_date, tomorrow;

tomorrow.set(11, 19);

due_date = tomorrow;
```

Program 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
  - Storin

Display 10.5 (1) nte

Display 10.5 (3)

Display 10.5 (2)

Display 10.5 (4)

The BankAccount Class (*part 1 of 4*)

```
//Program to demonstrate the class BankAccount.
#include <iostream>
using namespace std;

//Class for a bank account:
class BankAccount
{
public:
    void set(int dollars, int cents, double rate);
    //Postcondition: The account balance has been set to $dollars.cents;
    //The interest rate has been set to rate percent.

    void set(int dollars, double rate);
    //Postcondition: The account balance has been set to $dollars.00.
    //The interest rate has been set to rate percent.

    void update();
    //Postcondition: One year of simple interest has been
    //added to the account balance.

    double get_balance();
    //Returns the current account balance.

    double get_rate();
    //Returns the current account interest rate as a percentage.

    void output(ostream& outs);
    //Precondition: If outs is a file output stream, then
    //outs has already been connected to a file.
    //Postcondition: Account balance and interest rate have been written to the
    //stream outs.
private:
    double balance;
    double interest_rate;

    double fraction(double percent);
    //Converts a percentage to a fraction. For example, fraction(50.3) returns 0.503.
};

int main()
{
    BankAccount account1, account2;
    cout << "Start of Test:\n";
```

The member function set *is overloaded.*

Display 10.5
(1/4)

The BankAccount Class (*part 2 of 4*)

```
    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;
}

void BankAccount::set(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }

    balance = dollars + 0.01*cents;
    interest_rate = rate;
}

void BankAccount::set(int dollars, double rate)
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }

    balance = dollars;
    interest_rate = rate;
}
```

Calls to the overloaded member function set

Definitions of overloaded member function set

Display 10.5 (2/4)

Display 10.5 (3/4)

The BankAccount Class (*part 3 of 4*)

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

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

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

double BankAccount::get_rate()
{
    return interest_rate;
}

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

In the definition of a member function, you call another member function like this.

Stream parameter that can be replaced with either cout or with a file output stream

Display 10.5
(4/4)

The BankAccount Class (*part 4 of 4*)

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%
```

Calling Public Members

- Recall that if calling a member function from the main function of a program, you must include the object name:

```
account1.update( );
```

Calling Private Members

- When a member function calls a private member function, an object name is not used
  - `fraction (double percent);` is a private member of the `BankAccount` class
  - `fraction` is called by member function `update`

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

- Objects of classes can be used as formal parameters of a function

```
void update(BankAccount& old)
{
   old.update();
}
```

Another example from Display 10.5

```
//Uses iostream:
void BankAccount::output(ostream& outs)
{
    outs.setf(ios::fixed);
    outs.setf(ios::showpoint);
    outs.precision(2);
    outs << "Account balance $" << balance << endl;
    outs << "Interest rate " << interest_rate << "%" << endl;
}
int main( )
{
    BankAccount account1(100, 2.3), account2;
     account1.output(cout);
}
```

Another example from Lab04_template.cpp

```
void CDAccount::input(istream& inStream)
{
   inStream >> balance;
   inStream >> interestRate;
   inStream >> term;
}
int main()
{
   double balance, intRate, int term;
   CDAccount account = CDAccount( 100.0, 10.0, 6 );
   account.output(cout);
   cout << "Enter CD initial balance, interest rate, " << " and
   term: " << endl;
   account.input(cin);
```

- A function may return an object, i.e., the return type of a function can be a class

```
BankAccout new_account(BankAccount old)
{
    BankAccount temp;
    temp.set(0, old.get_rate());
    return temp;
}

BankAccount a;
a = new_account(old_account);
```

Constructors

- A **constructor** can be used to initialize member variables when an object is declared
  - A constructor is a member function that is usually 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, is used in declaring or defining a constructor

Constructor Declaration

- A constructor for the BankAccount class could be declared as:

```
class BankAccount
{
    public:
       BankAccount(int dollars, int cents, double
rate);
        //initializes the balance to $dollars.cents
        //initializes the interest rate to rate
percent

        …//The rest of the BankAccount definition
 };
```

Constructor Definition

- The constructor for the BankAccount class could be defined as

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

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

Overloading Constructors

- Constructors can be **overloaded** by defining constructors with different parameter lists
  - Other possible constructors for the BankAccount class might be

```
BankAccount (double balance, double interest_rate);

BankAccount (double balance);

BankAccount (double interest_rate);

BankAccount ( );
```

Calling A Constructor (1)

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

```
BankAccount  account1;

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

Calling A Constructor (2)

- A constructor is called in the object **declaration**

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

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

Or…

```
BankAccount account1;
account1 = BankAccount(999, 99, 5.5);
//another object is created.
// For another example, see demo code.
//This is like an ordinary function call learned in CS1.
```

Or…

```
BankAccount * account1Ptr;
account1Ptr = new BankAccount(999, 99, 5.5);
// For another example, see demo code.
```

The Default Constructor

- A default constructor uses no parameters
- A default constructor for the BankAccount class could be declared in this way

```
class BankAccount
{
 public:
      BankAccount( );
      // initializes balance  to $0.00
      // initializes rate to 0.0%
         … // The rest of the class
definition
};
```

Default Constructor Definition

- The default constructor for the BankAccount class could be defined as

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

- It is a good idea to always include a default constructor even if you do not want to initialize variables

Calling the Default Constructor

- The default constructor is called during declaration of an object
- An argument list is NOT used

```
BankAccount  account1;
//Correct.
 //Uses the default BankAccount constructor
 //when declaring an object.


 BankAccount  account1;
 acount1 = BankAccount();
 //Correct.
//Uses the default BankAccount constructor explicitly,
//when doing an assignment. For another example, see demo code.

 BankAccount account1( );
 //Does not correctly use the default constructor.
//In fact, the compiler thinks that this is a
//function declaration!
```

Default Constructors

- If your program does not provide any constructor for a class defined by you, C++ generates a default one for you that does nothing.
- If your program does provide some constructor (maybe only one), but no default constructor, C++ does NOT generate a default one.

See the demo program...

Initialization Sections

- An **initialization section** in a function definition provides an alternative way to initialize member variables

```
BankAccount::BankAccount( ): balance(0),
                             interest_rate(0.0);

{
   // No code needed in this example
}
```

  - The values in parenthesis are the initial values for the member variables listed

Parameters and Initialization

- Member functions with parameters can use initialization sections

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

  - Notice that the parameters can be arguments in the initialization

Section 10.2 Exercises

- Can you
  - Describe the difference between a class and a structure?
  - Explain why member variables are usually private?
  - Describe the purpose and usage of a constructor?
  - Use an initialization section in a function definition?

10.3

Abstract Data Types

Abstract Data Types

- A data type consists of a collection of values together with a set of basic operations defined on the values
  - example: int type and its associated valid operations
- A data type is an **Abstract Data Type (ADT)** if programmers using the type do not have access to the details of how the values and operations are implemented
  - example: int, double

Classes To Produce ADTs

- To define a class so it is an ADT
  - Separate the specification of how the type is used by a programmer from the details of how the type is implemented
  - Make all member variables **private** members
  - Helper functions should be private members
  - Basic operations a programmer needs should be **public** member functions
  - Fully specify how to use each public function

ADT Interface

- The ADT interface tells how to use the ADT in a program
  - The interface consists of
    - The public member functions' declarations or prototypes
    - The comments that explain how to use those functions
  - The interface should be all that is needed to know how to use the ADT in a program

ADT Implementation

- The ADT implementation tells how the interface is **realized** in C++
  - The **implementation** consists of
    - The private members of the class
    - The definitions of public and private member functions
  - The implementation of a class's interface is needed to run a program that uses the class.
  - The implementation is not needed to write the main part of a program or any non-member functions

ADT Benefits

- Changing an ADT implementation does not require changing a program that uses the ADT
- ADT's make it easier to divide work among different programmers
  - One or more can write the ADT
  - One or more can write code that uses the ADT
- Writing and using ADTs breaks the larger programming task into smaller tasks

Program Example
The BankAccount ADT

- In the version of the `BankAccount` ADT shown in Display 10.7.
  - Data is stored as three member variables
    - The dollars part of the account balance
    - The cents part of the account balance
    - The interest rate
  - This version stores the interest rate as a fraction
  - The public portion of the class definition remains unchanged from the version of Display 10.6

Same interface, different implementation

Display 10.6 Display 10.7

Display 10.6
(1/3)

DISPLAY 10.6 Class with Constructors *(part 1 of 3)*

```
1  //Program to demonstrate the class BankAccount.
2  #include <iostream>
3  using namespace std;
4  //Class for a bank account:
5  class BankAccount
6  {
7  public:
8      BankAccount(int dollars, int cents, double rate);
9      //Initializes the account balance to $dollars.cents and
10     //initializes the interest rate to rate percent.
11     BankAccount(int dollars, double rate);
12     //Initializes the account balance to $dollars.00 and
13     //initializes the interest rate to rate percent.
14     BankAccount( );
15     //Initializes the account balance to $0.00 and the interest rate to 0.0%.
```

This definition of BankAccount *is an improved version of the class* BankAccount *given in Display 10.5.*

Default constructor

(continued)

DISPLAY 10.6 Class with Constructors *(part 2 of 3)*

```
16      void update( );
17      //Postcondition: One year of simple interest has been added to the account
18      //balance.
19      double get_balance( );
20      //Returns the current account balance.
21      double get_rate( );
22      //Returns the current account interest rate as a percentage.
23      void output(ostream& outs);
24      //Precondition: If outs is a file output stream, then
25      //outs has already been connected to a file.
26      //Postcondition: Account balance and interest rate have been written to the
27      //stream outs.
28  private:
29      double balance;
30      double interest_rate;
31      double fraction(double percent);
32      //Converts a percentage to a fraction. For example, fraction(50.3)
33      //returns 0.503.
34  };
35
36  int main( )
37  {
38      BankAccount account1(100, 2.3), account2;

39      cout << "account1 initialized as follows:\n";
40      account1.output(cout);
41      cout << "account2 initialized as follows:\n";
42      account2.output(cout);
43      account1 = BankAccount(999, 99, 5.5);
44      cout << "account1 reset to the following:\n";
45      account1.output(cout);
46      return 0;
47  }
48
49  BankAccount::BankAccount(int dollars, int cents, double rate)
50  {
51      if ((dollars < 0) || (cents < 0) || (rate < 0))
52      {
53          cout << "Illegal values for money or interest rate.\n";
54          exit(1);
55      }
```

This declaration causes a call to the default constructor. Notice that there are no parentheses.

An explicit call to the constructor `BankAccount::BankAccount`

(continued)

Display 10.6 (2/3)

Display 10.6 (3/3)

DISPLAY 10.6 Class with Constructors *(part 3 of 3)*

```
56      balance = dollars + 0.01*cents;
57      interest_rate = rate;
58  }
59
60  BankAccount::BankAccount(int dollars, double rate)
61  {
62      if ((dollars < 0) || (rate < 0))
63      {
64          cout << "Illegal values for money or interest rate.\n";
65          exit(1);
66      }
67      balance = dollars;
68      interest_rate = rate;
69  }
70
71  BankAccount::BankAccount( ) : balance(0), interest_rate(0.0)
72  {
73     //Body intentionally empty
74  }
```

<Definitions of the other member functions are the same as in Display 10.5.>

Screen Output

```
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%
```

DISPLAY 10.7 Alternative BankAccount Class Implementation *(part 1 of 3)*

```
1  //Demonstrates an alternative implementation of the class BankAccount.
2  #include <iostream>
3  #include <cmath>
4  using namespace std;
5  //Class for a bank account:
6  class BankAccount
7  {
8  public:
9      BankAccount(int dollars, int cents, double rate);
10     //Initializes the account balance to $dollars.cents and
11     //initializes the interest rate to rate percent.
12     BankAccount(int dollars, double rate);
13     //Initializes the account balance to $dollars.00 and
14     //initializes the interest rate to rate percent.
15     BankAccount( );
16     //Initializes the account balance to $0.00 and the interest rate to 0.0%.
17     void update( );
18     //Postcondition: One year of simple interest has been added to the account
19     //balance.
20     double get_balance( );
21     //Returns the current account balance.
22     double get_rate( );
23     //Returns the current account interest rate as a percentage.
24     void output(ostream& outs);
25     //Precondition: If outs is a file output stream, then
26     //outs has already been connected to a file.
27     //Postcondition: Account balance and interest rate
28     //have been written to the stream outs.
29 private:
30     int dollars_part;
31     int cents_part;
32     double interest_rate;//expressed as a fraction, for example, 0.057 for 5.7
33     double fraction(double percent);
34     //Converts a percentage to a fraction. For example, fraction(50.3)
35     //returns 0.503.
36     double percent(double fraction_value);
37     //Converts a fraction to a percentage. For example, percent(0.503)
38     //returns 50.3.
39 };
```

Notice that the public members of BankAccount *look and behave exactly the same as in Display 10.6.*

New

(continued)

Display 10.7 (1/3)

DISPLAY 10.7 Alternative BankAccount Class Implementation *(part 2 of 3)*

```
40  int main( )
41  {
42      BankAccount account1(100, 2.3), account2;
43
44      cout << "account1 initialized as follows:\n";
45      account1.output(cout);
46      cout << "account2 initialized as follows:\n";
47      account2.output(cout);
48
49      account1 = BankAccount(999, 99, 5.5);
50      cout << "account1 reset to the following:\n";
51      account1.output(cout);
52      return 0;
53  }
54
55  BankAccount::BankAccount(int dollars, int cents, double rate)
56  {
57      if ((dollars < 0) || (cents < 0) || (rate < 0))
58      {
59          cout << "Illegal values for money or interest rate.\n";
60          exit(1);
61      }
62      dollars_part = dollars;
63      cents_part = cents;
64      interest_rate = fraction(rate);
65  }
66
67  BankAccount::BankAccount(int dollars, double rate)
68  {
69      if ((dollars < 0) || (rate < 0))
70      {
71          cout << "Illegal values for money or interest rate.\n";
72          exit(1);
73      }
74      dollars_part = dollars;
75      cents_part = 0;
76      interest_rate = fraction(rate);
77  }
78
79  BankAccount::BankAccount( ) : dollars_part(0), cents_part(0), interest_rate(0.0)
80  {
81      //Body intentionally empty.
82  }
83
```

Since the body of `main` *is identical to that in Display 10.6, the screen output is also identical to that in Display 10.6.*

In the old implementation of this ADT, the private member function `fraction` *was used in the definition of* `update`*. In this implementation,* `fraction` *is instead used in the definition of constructors.*

(continued)

Display 10.7 (2/3)

DISPLAY 10.7 Alternative BankAccount Class Implementation *(part 3 of 3)*

```
84  double BankAccount::fraction(double percent_value)
85  {
86      return (percent_value/100.0);
87  }
88
89  //Uses cmath:
90  void BankAccount::update( )
91  {
92      double balance = get_balance( );
93      balance = balance + interest_rate*balance;
94      dollars_part = floor(balance);
95      cents_part = floor((balance - dollars_part)*100);
96  }
97
98  double BankAccount::get_balance( )
99  {
100     return (dollars_part + 0.01*cents_part);
101 }
102
103 double BankAccount::percent(double fraction_value)
104 {
105     return (fraction_value*100);
106 }
107
108 double BankAccount::get_rate( )
109 {
110     return percent(interest_rate);
111 }
112
113 //Uses iostream:
114 void BankAccount::output(ostream& outs)
115 {
116     outs.setf(ios::fixed);
117     outs.setf(ios::showpoint);
118     outs.precision(2);
119     outs << "Account balance $" << get_balance( ) << endl;
120     outs << "Interest rate " << get_rate( ) << "%" << endl;
121 }
```

The new definitions of `get_balance` *and* `get_rate` *ensure that the output will still be in the correct units.*

Display 10.7 (3/3)

Interface Preservation

- To preserve the interface of an ADT so that programs using it do not need to be changed
  - Public member declarations cannot be changed
  - Public member **definitions** (i.e., **implementation** or realization) can be changed
  - Private member functions can be added, deleted, or changed

Information Hiding

- Information hiding was referred to earlier as writing functions so they can be used like black boxes
- ADT's does information hiding because
  - The interface is all that is needed to use the ADT
  - Implementation details of the ADT are not needed to know how to use the ADT
  - Implementation details of the data values are not needed to know how to use the ADT

Section 10.3 Exercises

- Can you
  - Describe an ADT?
  - Describe how to implement an ADT in C++?
  - Define the interface of an ADT?
  - Define the implementation of an ADT?

10.4

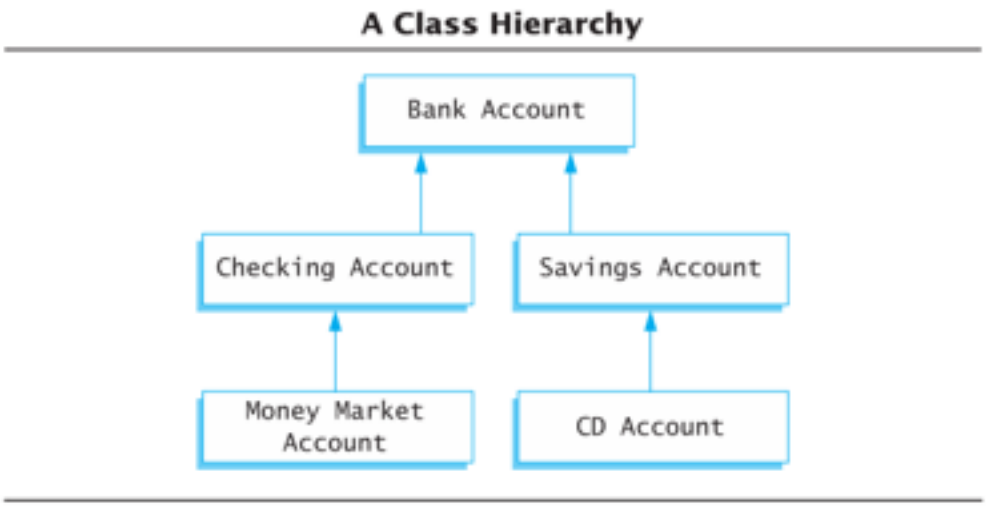
Introduction to Inheritance

Inheritance

- Inheritance refers to derived classes
  - Derived classes are obtained from another class by adding features
  - A derived class inherits the member functions and variables from its parent class without having to re-write them

Inheritance Example

- Natural hierarchy of bank accounts
- Most general: A Bank Account stores a balance
- A Checking Account "IS A" Bank Account that allows customers to write checks
- A Savings Account "IS A" Bank Account without checks but higher interest

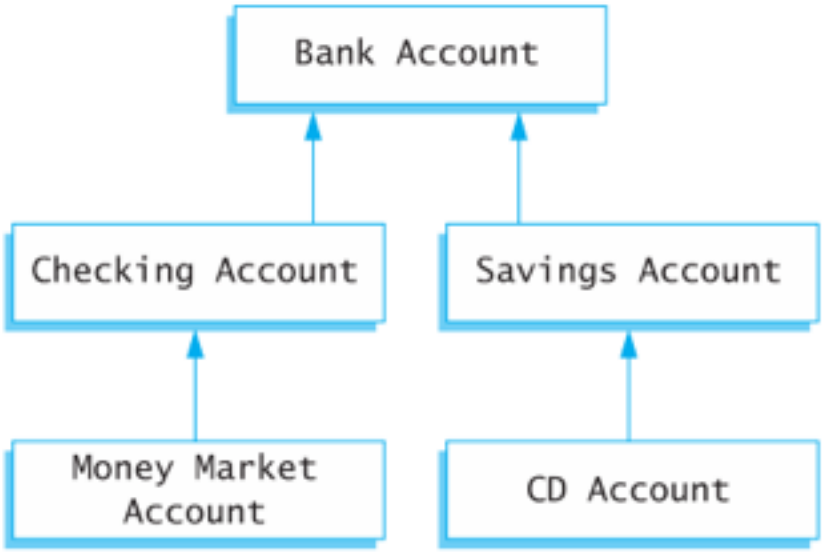


Accounts are more specific as we go down the hierarchy

Each box can be a class

Display 10.8

A Class Hierarchy



Inheritance Relationships

- The more specific class is a **derived** or **child** class
- The more general class is the **base**, **super**, or **parent** class
- If class B is derived from class A
  - Class B is a derived class of class A
  - Class B is a child of class A
  - Class A is the parent of class B
  - Class B inherits the member functions and variables of class A

Define Derived Classes

- Give the class name as normal, but add a colon and then the name of the base class

```
class SavingsAccount : public BankAccount
{
  …
}
```

- Objects of type `SavingsAccount` can access member functions defined in `SavingsAccount` or `BankAccou`

Display 10.9 (1-3)

<Everything from Display 10.6 should be inserted here except for the `main` function.>

```
1   class SavingsAccount : public BankAccount
2   {
3   public:
4       SavingsAccount(int dollars, int cents, double rate);
5       //Other constructors would go here
6       void deposit(int dollars, int cents);
7       //Adds $dollars.cents to the account balance
8       void withdraw(int dollars, int cents);
9       //Subtracts $dollars.cents from the account balance
10  private:
11  };
12  int main( )
13  {
14      SavingsAccount account(100, 50, 5.5);
15      account.output(cout);
16      cout << endl;
17      cout << "Depositing $10.25." << endl;
18      account.deposit(10,25);
19      account.output(cout);
20      cout << endl;
21      cout << "Withdrawing $11.80." << endl;
22      account.withdraw(11,80);
23      account.output(cout);
24      cout << endl;
25      return 0;
26  }
```

The colon indicates that the class `SavingsAccount` *is derived from the class* `BankAccount`

Only new member functions or variables need to be defined

Display 10.9 (1/3)

Display 10.9 (2/3)

The `SavingsAccount` *constructor invokes the* `BankAccount` *constructor. Note the preceding colon.*

```
27  SavingsAccount::SavingsAccount(int dollars, int cents, double rate):
28      BankAccount(dollars, cents, rate)
29  {
30      //deliberately empty
31  }

32  void SavingsAccount::deposit(int dollars, int cents)
33  {
34      double balance = get_balance();
35      balance += dollars;
36      balance += (static_cast<double>(cents) / 100);
37      int new_dollars = static_cast<int>(balance);
38      int new_cents = static_cast<int>((balance - new_dollars) * 100);
```

The `deposit` *function adds the new amount to the balance and changes the member variables via the* `set` *function*

For more information on type casting,
http://www.cplusplus.com/doc/tutorial/typecasting/

```
39          set(new_dollars, new_cents, get_rate());
40      }

41      void SavingsAccount::withdraw(int dollars, int cents)
42      {
43          double balance = get_balance();
44          balance -= dollars;
45          balance -= (static_cast<double>(cents) / 100);
46          int new_dollars = static_cast<int>(balance);
47          int new_cents = static_cast<int>((balance - new_dollars) * 100);
48          set(new_dollars, new_cents, get_rate());
49      }
```

The `withdraw` *function subtracts the amount from the balance and changes the member variables via the* `set` *function*

Screen Output

```
Account balance $100.50
Interest rate 5.50%
Depositing $10.25.
Account balance $110.75
Interest rate 5.50%
Withdrawing $11.80.
Account balance $98.95
Interest rate 5.50%
```

Display 10.9(3/3)

Section 10.4 Exercises

- Can you
  - Define object?
  - Define class?
  - Describe the relationship between parent and child classes?
  - Describe the benefit of inheritance?