Structures and Classes in C++

CS 16: Solving Problems with Computers I
Lecture #17

Ziad Matni
Dept. of Computer Science, UCSB

Announcements

- Lab #9 is due on the last day of classes: **Friday, 12/2**
- If you do not have a lab partner, you MUST see me after class today (your only chance)
- Homework #16 is due on Thursday, 12/1
  - **NO LATE SUBMISSIONS WILL BE ALLOWED FOR HW 16!**
  - I will post solutions to HW15 and HW16 after Thursday class

Lecture Outline

CH. 10

- Structures
- Classes

Remaining To-Dos

M	T	W	Th	F
11/28	**11/29** **HW #15 due** *Structures & Classes*	**11/30**	**12/1** **HW #16 due** *Review for Final Exam*	**12/2** **LAB #9 due**
12/5	**12/6** **FINAL EXAM At 4 PM**			

Structures

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

- Some pre-defined classes you have used are:
  - **ifstream**
  - **string**

- You can define your own classes as well

Class Definitions

- To define a "class", we need to…
  - Describe the kinds of values the variable can hold
    - Numbers? Characters? Both? Others?
  - Describe the member functions
    - What can we do with these values?

- We will start by defining ***structures*** as a first step toward defining classes

Structures

- A structure can be viewed as an **object**

- Let’s say it does not contain any member functions (for now…)

- It does contain multiple values of possibly different types

- We’ll call these **member variables**

Structures

- These multiple values are logically related to one another and come together as a single item
  - Examples:
    A bank Certificate of Deposit (CD) which has the following values:

    a balance
    an interest rate
    a term (how many months to maturity)

    What kind of values should these be?!

  - A student record which has the following values:

    the student's ID number
    the student's last name
    the student's first name
    the student's GPA

    What kind of values should these be?!

The CD Structure Example: Definition

- The Certificate of Deposit structure can be defined as

```
struct CDAccount
{
    double balance;
    double interest_rate;
    int term;
} ;
```

Remember this semicolon!

- Keyword **struct** begins a structure definition
- **CDAccount** is the structure ***tag*** – this is the structure's **type**
- Member names are ***identifiers*** declared in the braces

Using the Structure

- Structure **definition** should be 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

- **Structure Value** consists of all the values of the member variables

- 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

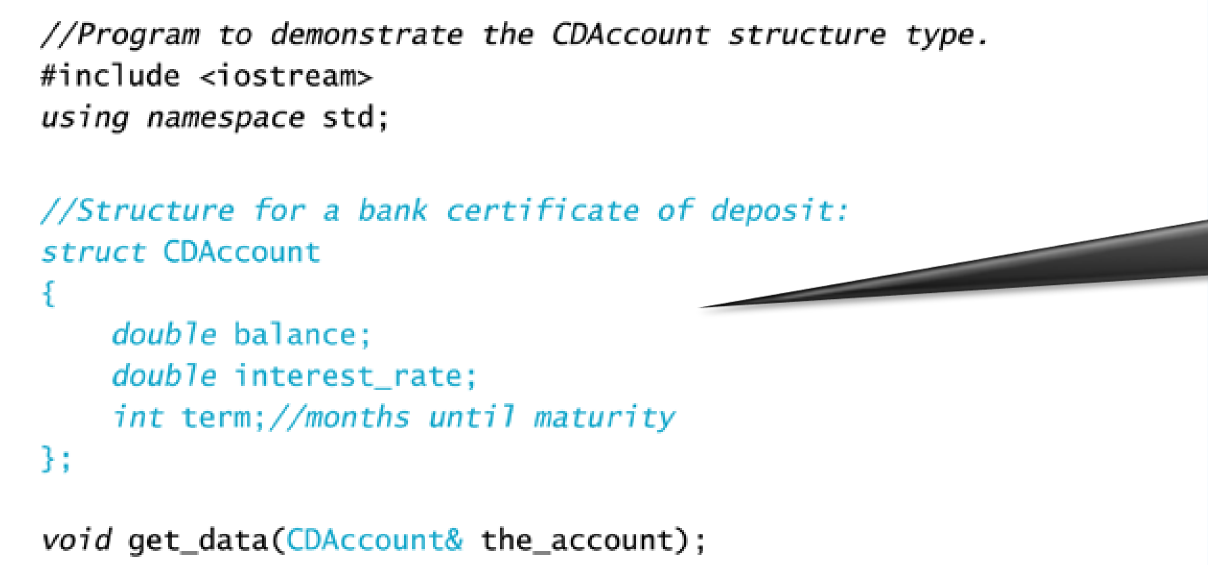
- Syntax to specify a member variable (note the '**.**')
  Structure_Variable_Name **.** *Member_Variable_Name*

  - Given the declaration:

    ```
    CDAccount  my_account, your_account;
    ```

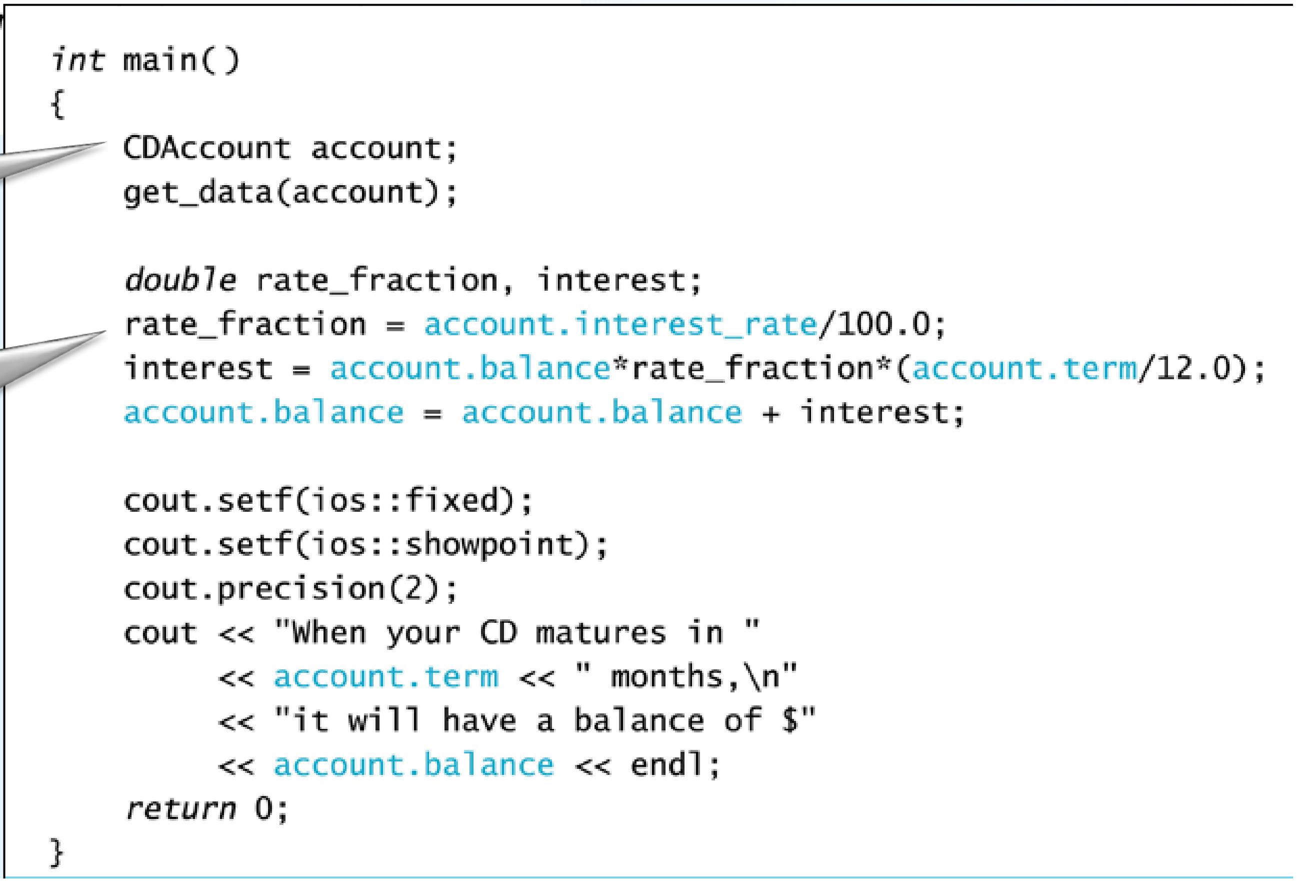
  - Use the **dot operator** to specify a member variable

    ```
    my_account.balance
    my_account.interest_rate
    my_account.term
    ```



```
//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.bala
//have been given values that the
```

Note the struct definition is placed before main()

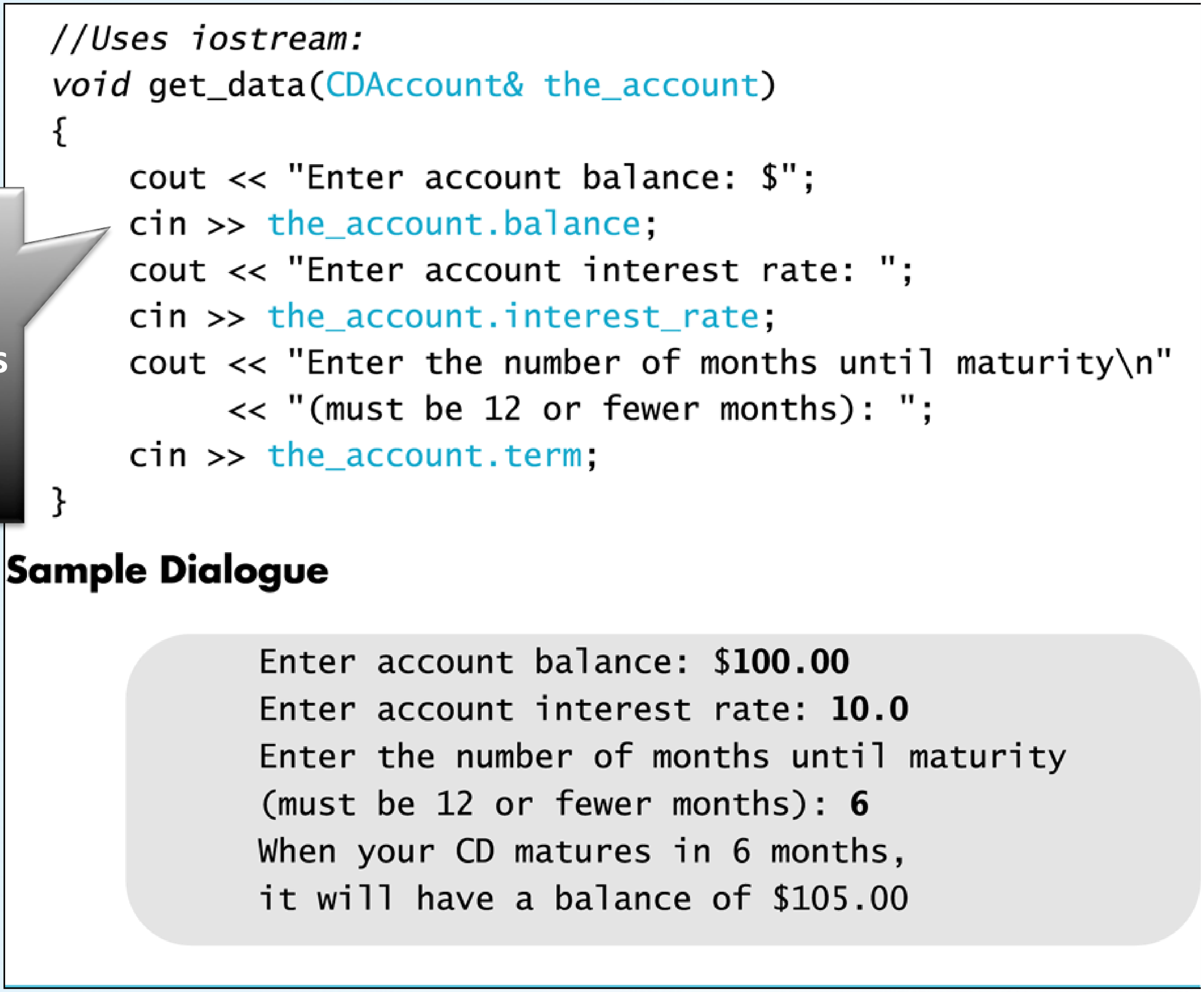


```
int main()
{
    CDAccount account;
    get_data(account);

    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;
}
```

Note the declaration of CDAccount

Note the calculations done with the structure's member variables

Note the use of the structure's member variables with an input stream



```
//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
```

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 either **call-by-value** or **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

Structures as Return Types

- Structures can also 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;
}
```

What is this function doing?

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);
```

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;
```

- Note: This last line assigns *all member variables* in **your_account** the corresponding values in **my_account**

Hierarchical Structures

- Structures can contain member variables that are also structures

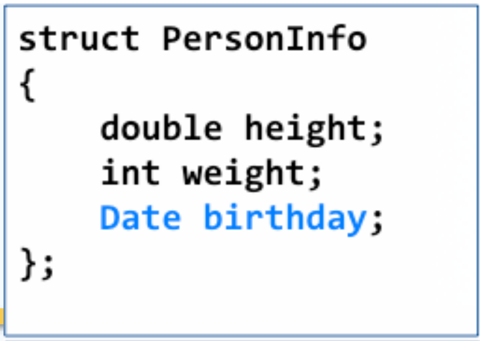
```
struct Date
{
    int month;
    int day;
    int year;
};
```

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

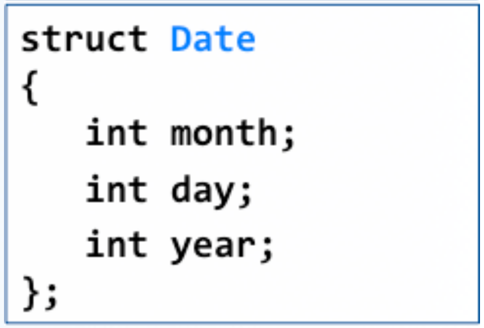
- struct **PersonInfo** contains a **Date** structure

Using **PersonInfo**

An example on . usage



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



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

- A variable of type **PersonInfo** is declared:

```
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};
```

Classes

Classes

- Reminder:
  A class is a data type whose variables are 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 variables

Main Differences: **structure** vs **class**

- Both *classes* and *structures* can have a mixture of public and private members and can have member functions
  - Although, often we'll leave functions for the *classes* and not the *structures*.
- *Structures* have default **public** members and *classes* have default **private** members.
  - More later on public vs private members…
- Classes may not be used when interfacing with C, because C does not have a concept of classes.

A Class Example

- Let’s create a new type called **DayOfYear** as a class
- First: 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
  - Here, we’ll use just one member function called **output**

Class **DayOfYear** Definition

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

Member Function **Declaration**

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
  - Note the use of the :: in the following

- 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***

- Indates *what class*
  a member function is a member of

- Example: **void DayOfYear::output( )** indicates that function output is a member of the **DayOfYear** class

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

‘::’ Operator vs. ‘.’ Operator

- ‘::’ is used with *classes* to identify a member

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

- ‘.’ is used with *variables* to identify a member

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

Calling Member Functions

- Calling the **DayOfYear** member function **output**:

```
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 function

- Also, note how similar this is to other class member functions call-outs that we've done, such as:

```
string Name = "Jimbo Jones";
int stlen = Name.length( );
```

Member Variables/Functions

Private vs. Public

- 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 tries to access a private member, the compiler will give an error message
  - Private is the default setting in classes

Private Variables

- Private variables **cannot** be accessed directly by the main program – only by other member functions of the class

- If we want the program to be able to change these variables' values, then they must be declared as **public** member functions of the class

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 called *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 called *public members* of the class

Example

```
class DayOfYear {
   public:
      void input();
      void output();
   private:
      void check_results();
      int var1, var2;
};
```

The member functions **input()** and **output()** are accessible from the **main()** or other functions in the program.

The member function **check_results()** is strictly to be used internally in **DayOfYear** class workings, as are int variables **var1** and **var2**.

Example from the Textbook *Display 10.4*

- The program takes in user input on today's date and compares it to J.S. Bach's birthday (i.e. a specific date of 3/21)

- Utilizes a user-defined class called **DayOfYear** which holds a date and a month, but ALSO does functions like:
  - Input date
  - Check date against set birthday
  - Outputs results

The **main()** function

```
int main () {
    DayOfYear today, bach_birthday;
    cout << “Enter today’s date:\n”;
    today.input();
    cout << “Today’s date is: ”;
    today.output();

    bach_birthday.set(3, 21);
    cout << “Bach’s Birthday is: ”;
    bach_birthday.output();

    if ((today.get_month() == bach_birthday.get_month()) &&
         (today.get_day() == bach_birthday.get_day()) {
        cout << “Happy Birthday, J.S. Bach!!!\n”; }

    return 0;
}
```

Note “today” & “bach_birthday” are both ***objects*** *of the* ***class*** *DayOfYear*

.input() and .output() are member functions of DayOfYear class. ***Must be public b/c main() is using them.***

.set() is a public member function too.

.get_month() and get_day() are public member functions too.

What variable types do they look like they return?

DayOfYear Class Definition

```
class DayOfYear {
   public:
      void input();
      void output();
      void set(int newmonth, int newday);
      int get_month();
      int get_day();
   private:
      void check_date();
      int month, day;
}
```

Q:
Why didn't we see this member function or these member variables in the main() part of the program?

*A: They're **private**!*

Define All The Member Functions…

input()

```
void input() {









}
```

STOP!!!

Define All The Member Functions…

input()

```
void DayOfYear::input() {

   cout “Enter the month as a number: ”;
   cin >> month;
   cout “Enter the day of the month: ”;
   cin >> day;

   check_date();
}
```

Calling a member function!

Is this
*a **private** or a **public** one?*

Define All The Member Functions…

output()

```
void DayOfYear::output() {

   cout “Month is: ”;
   cout << month << endl;
   cout “Day of the month is: ”;
   cout << day << endl;

}
```

Define All The Member Functions…

set(), ***get_month()*** and ***get_day()***

```
void DayOfYear::set(int newmonth, int newday) {
    month = newmonth;
    day = newday;
    check_date();
}

int DayOfYear::get_month() {
    return month;
}

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

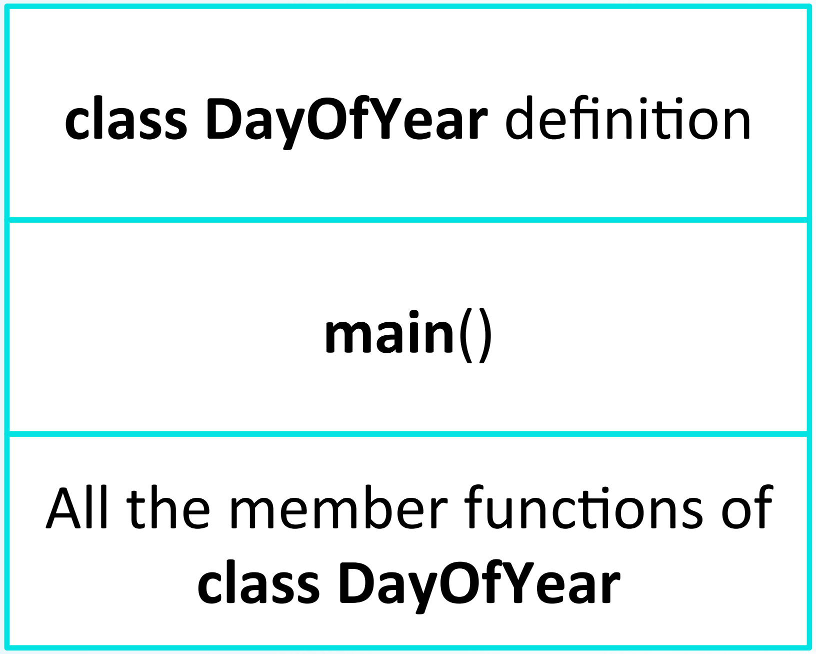
Define All The Member Functions…

check_date()

```
void DayOfYear::check_date() {
    if ( (month < 1) || (month > 12)
        || (day < 1) || (day > 31) ) {

        cout << “Illegal date. Aborting program!\n”;
        exit(1);
    }
}
```

Putting It All Together

- Check Display 10.4 Example in Textbook for full program.



class DayOfYear definition

main()

All the member functions of **class DayOfYear**

- Looks familiar?
- Same approach with defining functions in C++

Using Private Variables

- It is a practice norm to make all member *variables* **private**
  - Although, this is not strictly required…
  - Private variables require member functions to perform *all* changing and retrieving of values

- Functions that allow you to *obtain* the values of member variables are called **accessor** functions.
  - Example: **get_day** in class **DayOfYear**

- Functions that allow you to *also change* the values of member variables are called **mutator** functions.
  - Example: **set** in class **DayOfYear**

Review: Declaring an **Object**

- Once a **class** is defined, an **object** of the class is declared just as variables of any other type
  - This is similar to when you declare a structure in C++

- 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;
```

Review: Calling Public Members

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

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

- Again, just like when we used member functions of pre-defined classes, like `string`

Calling Private Members

- When a *member function* calls a **private** member function, an object name is not used

- Example: if `fraction (double percent);` is a private member of the class **BankAccount**
  - And if `fraction` is called by another member function, `update`

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

NOT: `BankAccount::fraction(interest_rate)*balance;`

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 and is automatically called when an object of the class is declared
  - RULE: 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

- Consider a class called **BankAccount**
- A constructor for the **BankAccount** class could be declared as follows:

```
 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

Calling A Constructor

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

```
BankAccount  account1;

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

Calling A Constructor

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

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

- This creates a **BankAccount** object and calls the constructor therein to initialize the member variables to 10, 50 and 2.0

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 ( );
```

The Default Constructor

- A default constructor uses no parameters and looks like this:

```
BankAccount( )
```

- 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
};
```

- **SEE EXAMPLE IN THE BOOK: Display 10.6**

Default Constructor Definition

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

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

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

Initialization Sections

- An initialization section in a function definition provides an ***alternative*** way (to the last slide) 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 *also* 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

To Dos

- Homework #16 for Thursday
  - LAST ONE! HURRAY!
  - No late submissions allowed
- Lab #9 due on Friday

</LECTURE>