Chapter 7.7 & 9
Defining Structs & Classes
What Is a Class?

- A class is a data type that is defined by a user to represent an object.
- Most of the data types we have used are built-in types, such as
  - int, char, float, double, long, char*
- We can define our own data types using
  - typedefs -> typedef type TypeName;
  - structs
  - classes
Class Definitions

- A class definition may include
  - Properties and functions that apply to the entire class [class variables, class methods]
  - Properties that are common to every member. [instance variables]
  - Functions that are available to every member. [methods or member functions]

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

7.7 Structures
7.8 Pointers to structs
8.1 Classes
10.3 Abstract Data Types
10.4 Introduction to Inheritance
10.1 Savitch

Structures
Structures (keyword struct)

- A struct can be viewed as an object
  - Used when multiple values are needed to describe an object. Examples?
  - Doesn't need to contain member functions (The structures used here have no member functions)
  - Contains multiple values of possibly different types
    - Multiple related values logically describe 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

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

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

Remember this semicolon!
Using the Structure

- Structure definition is generally placed outside any function definition in the global space.
  - 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
Local structure variables

Stack

my_account

<table>
<thead>
<tr>
<th>balance</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>interest_rate</td>
<td>?</td>
</tr>
<tr>
<td>term</td>
<td>?</td>
</tr>
</tbody>
</table>

your_account

<table>
<thead>
<tr>
<th>balance</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>interest_rate</td>
<td>?</td>
</tr>
<tr>
<td>term</td>
<td>?</td>
</tr>
</tbody>
</table>

The Value of a Structure
- Consists of the values of all 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:
    \[
    \text{Structure Variable Name}\_\text{Member Variable Name}
    \]

  - Given the declaration:
    \[
    \text{CDAccount my\_account, your\_account;}
    \]

    - Use the dot operator to specify a member variable
      \[
      \text{my\_account.balance}
      \]
      \[
      \text{my\_account.interest\_rate}
      \]
      \[
      \text{my\_account.term}
      \]
Member variables are used like any other variable of the same type.

```java
my_account.balance = 1000;
your_account.balance = 2500;
```

Notice that

```java
my_account.balance and your_account.balance are different variables!
```

```java
my_account.balance = my_account.balance + interest;
```
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;
}
A Structure Definition (part 2 of 2)

```cpp
//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**

```c
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 variables are scoped to their struct.
- Using duplicate names in different structure types is not a problem. No conflict here!

```c
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
  - The formal parameter can be a pointer (later).
- Example:
  ```c
  void get_data(CDAccount& the_account);
  ```
  - Uses the structure type `CDAccount` we saw earlier as the type for a call-by-reference parameter
  - Difference than a language like java where all object variables are references.
The assignment operator is used to assign values to structures:

```c
CDAccount my_account;
CDAccount 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:
```c
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;
}
```
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:

```c
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

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

- `struct PersonInfo` contains a `Date` structure

```c
struct PersonInfo
{
    double height;
    int weight;
    Date birthday;
};
```
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 Structs (or 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

Let’s do the following...

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

- Define a structure called EmployeeRecord that has the following fields:
  - A name
  - A wage rate - to hold a monetary value
  - Accrued vacation - in whole days
  - A wage status for hourly vs salaried

- What types do these fields have?
Struct EmployeeRecord

- Now we can definition the EmployeeRecord

```c
struct EmployeeRecord {
    string name; // first and last name
    double wage_rate; // hourly or annual rate
    int accrued_vacation; // in whole days
    char wage_status; // ‘H’ = hourly; ‘S’=annual
};
```
11.0 Gaddis

7.8 Pointers to Structures
Pointers to Structures

- A structure variable has an address
- Pointers to structures are variables that can hold the address of a structure:
  
  ```
  Student *studentPtr;
  ```

- Can use `&` operator to assign address:
  
  ```
  studentPtr = &student1;
  ```

- Structure pointer can be a function parameter
Accessing Structure Members via Pointer Variables

- **Must use () to dereference pointer variable, not field within structure:**
  
  ```cpp
  cout << (*studentPtr).studentID;
  ```

- **Can use structure pointer operator to eliminate () and use clearer notation:**
  
  ```cpp
  cout << studentPtr->studentID;
  ```
void getData(Student *s)
{
    // Get the student name.
    cout << "Student name: ";
    getline(cin, s->name);

    // Get the student ID number.
    cout << "Student ID Number: ";
    cin >> s->idNum;

    // Get the credit hours enrolled.
    cout << "Credit Hours Enrolled: ";
    cin >> s->creditHours;

    // Get the GPA.
    cout << "Current GPA: ";
    cin >> s->gpa;
}
Define the Student struct

- Can we define the student struct based on the previous slide?
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 like a structure definition except that members are private by default...
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;
};

Member Function Declaration
Struct vs Class – the true difference!

- **Structs** have **public** data members and functions by default.
  - Access struct members directly using the `.` (dot) operator:
    ```
    cout << bday.month << “/” << bday.year;
    ```

- **Classes** have **private** data members and functions by default.
  - Access class members using member functions called with the `.` (dot) operator
    ```
    bday.output();
    ```
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 (can be accessed by anyone).
- The keyword private identifies the members of a class that can be accessed only by member functions of the class (can only be accessed by class).
  - Members that follow the keyword private are private members of the class.
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` function.
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.
- Data members are directly accessible because member functions are always called from an object: `doy.output();`

```cpp
void DayOfYear::output()
{
    cout << "month = " << this->month
         << ", day = " << this->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 '.' operators

:: 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( );
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
6 class DayOfYear
7 {
8  public:
9    void output();  // Member function declaration
10    int month;
11    int day;
12  
13  int main()
14  {
15      DayOfYear today, birthday;
16      cout << "Enter today's date:\n";
17      cout << "Enter month as a number: ";
18      cin >> today.month;
19      cout << "Enter the day of the month: ";
20      cin >> today.day;
21      cout << "Enter your birthday:\n";
22      cout << "Enter month as a number: ";
23      cin >> birthday.month;
24      cout << "Enter the day of the month: ";
25      cin >> birthday.day;
26      cout << "Today's date is ";
27      today.output();
28      cout << "Your birthday is ";
29      birthday.output();
30      if (today.month == birthday.month
31          && today.day == birthday.day)
32          cout << "Happy Birthday!\n";
33      else
34          cout << "Happy Unbirthday!\n";
35      return 0;
36  }
37  //Uses iostream:
38  void DayOfYear::output()
39  {
40    cout << "month = " << month
41        << ", day = " << day << endl;
42  
(continued)
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!
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`.
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 four 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.
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

```cpp
void DayOfYear::set(int new_month, int new_day)
{
    month = new_month;
    day = new_day;
}
```
Public and 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  Class with Private Members (part 1 of 2)

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

(continued)
```cpp
cin >> month;  
cout << "Enter the day of the month: ";
cin >> day;  
check_date( );
}

void DayOfYear::output()
<The rest of the definition of DayOfYear::output is given in Display 10.3.>

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

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

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

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

**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!
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`.
- Member functions for
  - `set(int month, int day), input()`, `output()`
  - `get_month()` and `get_day()`
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`.
- Another term for Accessor is **getter** and Mutator is **setter**.
The Assignment Operator

- Objects and structures can be assigned values with the assignment operator (=) but cannot compare with operator (==)

  Example:
  ```plaintext
  DayOfYear due_date, tomorrow;

  tomorrow.set(11, 19);

  due_date = tomorrow;
  ```
Even more Ideal class

- Implement a member function that tells if one DayOfYear object is equal to another.
- Remember that to call a member function requires an object and the '.' operator.
  - What is the return type?
  - What is the parameter type?
  - What is the code?
Define isEqual for DayOfYear class
bool isEqual(DayOfYear doy)

bool DayOfYear::isEqual(DayOfYear doy)
{
    return (this->month == doy.get_month() &&
            this->day == doy.get_day());
}

// Why does this work?
if (date.isEqual(bachsBirthday))
    cout << "Today is Bach’s Birthday\n";
// C++ assigns this = &date (because of the ‘.’ operator
// in the function call above) behind the scenes.
General Class Definitions

- The syntax for a class definition is
  
  ```cpp
class Class_Name
{
  public:
    Member_Specification_1
    Member_Specification_2
    ...
    Member_Specification_n+1
  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:

  ```cpp
  class Cycle
  {
      // class definition lines
  }
  
  Cycle my_cycle, your_cycle;
  ```
Bicycle as a struct or class

```cpp
struct Cycle { // all members public
    int wheel_height;
    int num_wheels;
    int num_gears;
}; // separate functions to set values
Cycle initCycle(int wht, int nw, int ng) {
    Cycle tempBike;
    tempBike.wheel_height = wht;
    tempBike.num_wheels = nw;
    tempBike.num_gears = ng;
    return tempBike;
} // myBike and yourBike are distinct.
Bicycle myBike = initBicycle(15, 2, 10);
Bicycle yourBike = myBike;
```

```cpp
class Cycle { // public methods
    public:
        void set(int wht, int nw, int ng);
        ...
    private: // private members, functions
        int wheel_height;
        int num_wheels;
        int num_gears;
}; // member functions access fields directly
void Cycle::set(int wht, int nw, int ng) {
    wheel_height = wht;
    num_wheels = nw;
    num_gears = ng;
}
Bicycle myBike.set(15, 2, 10);
```
Constructors

- Member function that is automatically called when an object is created
- Purpose is to construct an object
- Constructor function name is class name
- Has no return type
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...
Consider a rectangle class

- What is needed to describe a rectangle?
- What functions can we do with a rectangle?
- Consider using a rectangle in math class?
class Rectangle {
    public:
    int getHeight() const;
    int getWidth() const;
    int calcPerimeter() const;
    int calcArea() const;
    void set(int h, int w);
    void addTo(int hDim, int wDim);
    void draw();

    private:
    int height;
    int width;
};
Default Constructors

- A default constructor is a constructor that takes no arguments.

- If you write a class with no constructor at all, C++ will write a default constructor for you, one that does nothing.

- A simple instantiation of a class (with no arguments) calls the default constructor:

  ```cpp
  Rectangle r;
  ```
Contents of Rectangle.h (Version 3)

1  // Specification file for the Rectangle class
2  // This version has a constructor.
3  #ifndef RECTANGLE_H
4  #define RECTANGLE_H
5
6  class Rectangle
7  {
8    private:
9        double width;
10        double length;
11    public:
12        Rectangle();          // Constructor
13        void setWidth(double);
14        void setLength(double);
15
16        double getWidth() const
17            { return width; }
18
19        double getLength() const
20            { return length; }
21
22        double getArea() const
23            { return width * length; }
24    };
25  #endif
Contents of Rectangle.cpp (Version 3)

1      // Implementation file for the Rectangle class.
2      // This version has a constructor.
3      #include "Rectangle.h"   // Needed for the Rectangle class
4      #include <iostream>      // Needed for cout
5      #include <cstdlib>       // Needed for the exit function
6      using namespace std;
7
8      /******************************************************************************
9      // The constructor initializes width and length to 0.0.       *
10     /******************************************************************************

11 Rectangle::Rectangle()
12 {
13     width = 0.0;
14     length = 0.0;
15 }
Contents of Rectangle.cc  Version3

```cpp
17
18  //*****************************************************************
19  // setWidth sets the value of the member variable width. *
20  //*****************************************************************
21
22  void Rectangle::setWidth(double w)
23  {
24      if (w >= 0)
25          width = w;
26      else
27          {
28          cout << "Invalid width\n";
29          exit(EXIT_FAILURE);
30          }
31  }
32  
33  //*****************************************************************
34  // setLength sets the value of the member variable length. *
35  //*****************************************************************
36
37  void Rectangle::setLength(double len)
38  {
39      if (len >= 0)
40          length = len;
41      else
42          {
43          cout << "Invalid length\n";
44          exit(EXIT_FAILURE);
45          }
46  }
```
(continued)
Program 13-7

```cpp
#include <iostream>
#include "Rectangle.h" // Needed for Rectangle class
using namespace std;

int main()
{
    Rectangle box; // Define an instance of the Rectangle class

    // Display the rectangle's data.
    cout << "Here is the rectangle's data:\n";
    cout << "Width: " << box.getWidth() << endl;
    cout << "Length: " << box.getLength() << endl;
    cout << "Area: " << box.getArea() << endl;
    return 0;
}
```

Program Output

Here is the rectangle's data:

Width: 0
Length: 0
Area: 0
Structs and classes

- When do we use structs? When do we use classes?
  - Mostly, we use classes but...
  - Structs are useful when code has to be accessible to both c and c++.
  - Structs are useful to describe an object that is internal to a class or if direct access to data members is required.
  - But even then, it’s probably better to just make it a class.
Access Specifiers

- Used to control access to members of the class

  - **public**: can be accessed by functions outside of the class

  - **private**: can only be called by or accessed by functions that are members of the class
More on Access Specifiers

- Can be listed in any order in a class
- Can appear multiple times in a class
- If not specified, the default is `private`
13.8

Passing Arguments to Constructors
Passing Arguments to Constructors

- To create a constructor that takes arguments:
  - indicate parameters in prototype:
    ```
    Rectangle(double, double);
    ```
  - Use parameters in the definition:
    ```
    Rectangle::Rectangle(double w, double len)
    {
        width = w;
        length = len;
    }
    ```
Passing Arguments to Constructors

- You can pass arguments to the constructor when you create an object:

  Rectangle r(10, 5);
Classes with No Default Constructor

- When all of a class's constructors require arguments, then the class has NO default constructor.

- When this is the case, you must pass the required arguments to the constructor when creating an object.
13.10

Overloading Constructors
Overloading Constructors

- A class can have more than one constructor

- Overloaded constructors in a class must have different parameter lists:

  ```
  Rectangle();
  Rectangle(double);
  Rectangle(double, double);
  ```
More About Default Constructors

- If all of a constructor's parameters have default arguments, then it is a default constructor. Done in declaration. For example:

  ```
  Rectangle(double w= 0, double l= 0);
  ```

- Creating an object and passing no arguments will cause this constructor to execute:

  ```
  Rectangle r;
  ```
Initialization Section

- C++ constructors have a special section that can be used to initialize data members:

```cpp
Rectangle::Rectangle(int w, int l) :
  width(w), length(l)
{
  // purposely left empty
}
```
Simple InventoryItem

- A simple object to describe an InventoryItem would include:
  - A text description of the item.
  - A cost of the item.
    - A price might also be included (not in book).
  - The number of units.
    - A SKU (stock keeping unit) for the item.
// This class has overloaded constructors.
#ifndef INVENTORYITEM_H
#define INVENTORYITEM_H
#include <string>
using namespace std;

class InventoryItem
{
private:
    string description; // The item description
    double cost;       // The item cost
    int units;         // Number of units on hand

public:
    // Constructor #1
    InventoryItem()
    { // Initialize description, cost, and units.
        description = "";
        cost = 0.0;
        units = 0; }

    // Constructor #2
    InventoryItem(string desc)
    { // Assign the value to description.
        description = desc;

        // Initialize cost and units.
        cost = 0.0;
        units = 0; }

    // Continued...
// Constructor #3
InventoryItem(string desc, double c, int u)  
  { // Assign values to description, cost, and units.
    description = desc;
    cost = c;
    units = u;  }

// Mutator functions
void setDescription(string d)  
  { description = d;  }

void setCost(double c)  
  { cost = c;  }

void setUnits(int u)  
  { units = u;  }

// Accessor functions
string getDescription() const  
  { return description;  }

double getCost() const  
  { return cost;  }

int getUnits() const  
  { return units;  }

};
Using `const` With Member Functions

- `const` appearing after the parentheses in a member function declaration specifies that the function will not change any data in the calling object.

```cpp
double getWidth() const;
double getLength() const;
double getArea() const;
```
Defining a Member Function

- When defining a member function:
  - Put prototype in class declaration
  - Define function using class name and scope resolution operator (::)

```cpp
int Rectangle::setWidth(double w)
{
    width = w;
}
```
Remember Accessors and Mutators

- **Mutator**: a member function that stores a value in a private member variable, or changes its value in some way.

- **Accessor**: function that retrieves a value from a private member variable. Accessors do not change an object's data, so they should be marked `const`.
13.3

Defining an Instance of a Class
Defining an Instance of a Class

- An object is an instance of a class
- Defined like structure variables:
  
  Rectangle r;

- Access members using dot operator:
  
  r.setWidth(5.2);
  
  cout << r.getWidth();

- Compiler error if attempt to access private member using dot operator
Program 13-1

```
1 // This program demonstrates a simple class.
2 #include <iostream>
3 using namespace std;
4
5 // Rectangle class declaration.
6 class Rectangle
7 {
8     private:
9         double width;
10         double length;
11     public:
12         void setWidth(double);
13         void setLength(double);
14         double getWidth() const;
15         double getLength() const;
16         double getArea() const;
17     };
18
19 //********************************************************************************
20 // setWidth assigns a value to the width member. *
21 //********************************************************************************
22
23 void Rectangle::setWidth(double w)
24 {
25     width = w;
26 }
27
28 //********************************************************************************
29 // setLength assigns a value to the length member. *
30 //********************************************************************************
31```
void Rectangle::setLength(double len)
{
    length = len;
}

/******************************************************
// getWidth returns the value in the width member. *
/******************************************************

double Rectangle::getWidth() const
{
    return width;
}

/******************************************************
// getLength returns the value in the length member. *
/******************************************************

double Rectangle::getLength() const
{
    return length;
}
double Rectangle::getArea() const
{
    return width * length;
}

int main()
{
    Rectangle box;    // Define an instance of the Rectangle class
    double rectWidth; // Local variable for width
    double rectLength; // Local variable for length

    // Get the rectangle's width and length from the user.
    cout << "This program will calculate the area of a\n";
    cout << "rectangle. What is the width? ";
    cin >> rectWidth;
    cout << "What is the length? ";
    cin >> rectLength;

    // Store the width and length of the rectangle
    // in the box object.
    box.setWidth(rectWidth);
    box.setLength(rectLength);
Program 13-1 (Continued)

// Display the rectangle's data.
cout << "Here is the rectangle's data:\n";
cout << "Width: " << box.getWidth() << endl;
cout << "Length: " << box.getLength() << endl;
cout << "Area: " << box.getArea() << endl;
return 0;

Program Output
This program will calculate the area of a rectangle. What is the width? 10 [Enter]
What is the length? 5 [Enter]
Here is the rectangle's data:
Width: 10
Length: 5
Area: 50
Avoiding Stale Data

- Some data is the result of a calculation.
- In the `Rectangle` class the area of a rectangle is calculated.
  - length x width
- If we were to use an `area` variable here in the `Rectangle` class, its value would be dependent on the length and the width.
- If we change `length` or `width` without updating `area`, then `area` would become stale.
- To avoid stale data, it is best to calculate the value of that data within a member function rather than store it in a variable.
3.11
Using Private Member Functions
Section 10.2 Exercises

Can you answer the following:

- 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 constructor?
- What is the purpose of a default constructor and when is one supplied automatically in a class?
Describe the difference between a class and a structure?
- structs are public by default; classes are private by default.

Explain why member variables are usually private?
- Protect programs from class changes and objects from internal corruption.

Describe purpose and usage of a constructor?
- Constructors are invoked when an object is created.
- Used to declare variables.
Answers to Exercises

- Use initialization section in a constructor?
  - (See slide 98, see slide 111)
    ```cpp
    DayOfYear::DayOfYear(int m, int d) : month(m), day(d) {} 
    
    DayOfYear::DayOfYear() : month(1), day(1) {} 
    ```

- What is the purpose and when is a default constructor provided automatically?
  - If a class does not provide a constructor, a default constructor will automatically be provided that does nothing.
  - A default constructor is needed to declare objects.
As practice consider the following

Take our DayOfYear class and add two constructors: one that takes the month and day to initialize both data members and a default constructor.

Even though, we already added a member function called isEquals, try it on your own. Remember it takes a DayOfYear object as parameter.

Add a member function called isAfter that compares the invoking object to an object passed in as a parameter. if the calling object is later in the year, then return true. Otherwise false.
class DayOfYear {
    public:
        int getMonth();
        int getDay();
        void output(ostream& outs);
        void input(istream& ins);
    private:
        void check_date();
        int month;
        int day;
};
Start with the class \texttt{DayOfYear}

- On the previous slide is the class \texttt{DayOfYear} before you make your changes.
- The slides that follow show changes
  - Add the two constructors \textbf{OR}
  - Add one constructor with default values.
    - But this is equivalent to 3 constructors:
      - \texttt{DayOfYear();} // default month = 1, day=1
      - \texttt{DayOfYear(int mon);} // default day = 1
      - \texttt{DayOfYear(int mon, int day);}
class DayOfYear {
    public:
        DayOfYear();
        DayOfYear(int mon, int day);
        int getMonth();
        int getDay();
        void output(ostream& outs);
        void input(istream& ins);
    private:
        void check_date();
        int month, day;
};
class DayOfYear {
    public:
        DayOfYear(int mon=1, int day=1);
        int getMonth();
        int getDay();
        void output(ostream& outs);
        void input(istream& ins);
    private:
        void check_date();
        int month, day;
};
Implementation of Constructor

// Set data members in initialization
// section, check_date() exits if not
// valid
DayOfYear::DayOfYear(int m, int d) :
month(m), day(d)
{
    check_date();
}

Not necessary to use initialization section!
Overloaded constructors

- Remember Rectangle class with the following constructors:
  - Rectangle();
  - Rectangle(double side);
  - Rectangle(double w, double l);
- What if we added?
  - Rectangle(double w); // Compiler error

Overloaded functions must have different number and/or types of parameters!!!
isAfter() member function

- It's tempting to pass an int month and int day, but let's compare to another DayOfYear!

// Add to public part of class
bool isAfter(DayOfYear d);

// Implementation outside of the class
bool DayOfYear::isAfter(DayOfYear d)
{
    return (getMonth() > d.getMonth()) ||
           (getMonth() == d.getMonth() &&
            getDay() > d.getDay());
}
Practice Finding the errors.

class Automobile {
    public:
        void setPrice(double price);
        void setProfit(double profit);
        double get_price();
    private:
        double get_profit();
        double price;
        double profit;
}; Automobile jaguar, hyundai;
hyundai.price = 4999.99; jaguar.set_price(59000.00);
double a_price = jaguar.get_price(), a_profit = hyundai.get_profit();
if (hyundai == jaguar)
    cout << "Want to swap cars?" << endl;
hyundai = jaguar;
See Errors

- If a data member or member function is private, it cannot be accessed outside of the class.
  - Errors are in red.
Practice Finding the errors.

class Automobile {
    public:
        void set_price(double price);
        void set_profit(double profit);
        double get_price();
    private:
        double get_profit();
        double price;
        double profit;
};
Automobile jaguar, hyundai;

hyundai.price=4999.99; jaquar.set_price(59,000.00);
double a_price = jaguar.get_price(), a_profit = hyundai.get_profit();
if (hyundai == jaguar) // No operator == exists for Automobile
    cout << “Want to swap cars?” << endl;
hyundai = jaguar;
11.0

Class parameters, const
Parameter passing efficiency

A call-by-value parameter less efficient than a call-by-reference parameter

- The parameter is a local variable initialized to the value of the argument
  - This results in two copies of the argument
  - For vectors and strings, it means dynamic allocation and copying the values. Expensive!!

A call-by-reference parameter is more efficient

- The parameter is a placeholder replaced by the argument
  - There is only one copy of the argument
Class Parameters

- It can be much more efficient to use call-by-reference parameters when the parameter is of a class type.
- When using a call-by-reference parameter:
  - If the function does not change the value of the parameter, mark the parameter so the compiler knows it should not be changed.
const Parameter Modifier

- To mark a call-by-reference parameter so it cannot be changed:
  - Use the modifier const before the parameter type
  - The parameter becomes a constant parameter
  - const used in the function declaration and definition
Passing structs as arguments

- Pass-by-values copies the struct member by member onto the stack into the parameters. - waste of time and space.
- Pass-by-reference copies only the address of the object onto the stack but the argument can be changed.
  - dangerous unless change is expected.
- We want an efficient way to pass objects (structs and classes) as arguments.
Struct as Function Arguments

- Using value parameter for structure can slow down a program, waste space with object copy
- Using a reference parameter will speed up program, but function may change data in structure
- Using a `const` reference parameter allows read-only access to reference parameter, does not waste space, speed

```cpp
void displayDate(const Date& date) {
    cout << date.month << ‘/’ <<
    date.day << ‘/’ << date.year;
}
```
Examples: Bank Account & Rectangle
Consider a BankAccount class

- First ask, what is it?
  - Answering “what is it?” gives us the data
  - Could be several data representations
    - E.g. a rational number could be...  
      - 3.1459 or 2/3 but is a fraction.

- Then ask, what can you do with it?
  - Answering “what can you do?” gives us the functions.
BankAccount

- What is it? All that it has or components.
  - Has A balance (but that balance can be expressed as...)
    - double balance; // Fractional cents problem
    - int cents; // More exact
    - int dollars, int cents; // Multiple components
  - Has An interest rate
  - Has An account number
    - unsigned long acctNumber;
    - string acctNumber; // Mix of chars, digits
BankAccount

- When a BankAccount is created, there is an initial deposit.
  - `BankAccount(int dollars = 0, int cents = 0);`
- Typical banking functions include:
  - `makeDeposit(double amount)`
  - `makeWithdrawl(double amount)`
  - `makeTransfer(double amnt, BankAccount ba)`
  - `addAccruedInterest(double rate)`
  - `issueMonthlyStatement()`
BankAccount con’t

- In order to issue a monthly statement, we need to keep track of transactions.
- An easy way to do that is to create a Transaction structure that is used by the BankAccount object.
- Private data for BankAccount includes:
  - double balance;
  - double interest_rate;
  - vector <Transaction> transactions;
struct transaction

- Every transaction has a dollar amount.
- Every transaction has a date (and time).
- Every transaction has a type.
  - We'll use a similar construct as wage_status in a previous example.
- There are 5 different transaction types:
  - Deposits
  - Transfers
  - Withdrawals
  - Overdraft
  - Interest payments
Struct transaction

- Here’s the struct that represents Transaction type in our BankAccount application.

```c
struct Transaction {
    char    type;         // 'D', 'W', 'O', 'I', 'T'
    double  amount;
    Time    time;         // Time struct or class
    string  src_account;  // consider transfers
};
```

- In order to issue a monthly statement, we need to keep a list of Transactions.
struct Transaction

- How do we keep track of transactions for the month?
  - `vector<Transaction> transactions;`
  - Write a method to shrink_wrap or initialize a transaction to be stored in `transactions`.

- How do we get a month’s worth of transactions?
  - Write a method to start at the end and go backwards until we hit the previous month.
Class BankAccount

- Remember that we want BankAccount to support normal banking functions such as:
  - `makeDeposit(double amount)`
  - `makeWithdrawl(double amount)`
  - `makeTransfer(double amnt, BankAccount& ba)`
  - `addAccruedInterest(double rate)`
  - `issueMonthlyStatement(DayOfYear from)`
  - `getBalance()`, `getInterestRate()`, `getAccountNum()`

- Each banking function will be represented by a member function.
Class BankAccount and struct Transaction

/*
* BankAccount class
*    keeps the data along with all of the functions that apply to a BankAccount.
*/
class BankAccount {
public:
    BankAccount(int dollars = 0, int cents = 0); // Includes default constructor

    void set(double balance, double interest_rate);
    double getBalance();
    double getInterestRate();
    unsigned long getAccountNum();

    // Public method interface
    double makeDeposit(double deposit);
    double makeWithdrawl(double withdrawal);
    double makeTransfer(double transfer, BankAccount toAcct);
    double addAccruedInterest();
}
Class BankAccount and struct Transaction

    string issueMonthlyStatement(DayOfYear from, DayOfYear until);

private:
// Constants that apply to the whole class
    static const char DEPOSIT = 'D';
    static const char WITHDRAWL = 'W';
    static const char INTEREST = 'I';
    static const char OVERDRAFT = 'O';
    static const char TRANSFER = 'T';

// Private utility methods
    Transaction shrink_wrap(char type, double amount, time_t date=0);
    void record_transaction(unsigned long account, char type, double amount, Date td);

// Instance variables
    unsigned long account_num;       // Accommodates a long number
    double balance;                  // A danger of fractional cents, see ADT
    double interest_rate;
    vector<Transaction> transactions; // Audit trail of all transactions.
};
C++ code for bank_account.cpp

A Rectangle needs to be...

- A shape with a positive, non-zero length and a height.

- A shape that can have equal length and height which makes it the special case of a Square.

- A shape that can be drawn needs a position and orientation (angle).
Rectangle

- What kinds of things do you do with a Rectangle?
  - Get length and height
  - Calculate Perimeter
  - Calculate Area
  - isSquare
  - hasSamePerimeter
  - hasSameArea
  - Draw, Move, Rotate... // drawing program.
Rectangle constructors

- Define a default constructor

- Define a constructor for the special case of square.

- Define a constructor for initializing both dimensions at creation.
class Rectangle {
    public:
        Rectangle(); //default constructor
        Rectangle(int side); // Square
        Rectangle(int length, int height);
        int get_length(); // length accessor
        int get_height(); // height accessor
        int calculatePerimeter();
        int calculateArea();
        bool hasSameArea(const Rectangle& r);
        bool hasSamePerimeter(const Rectangle& r);
        bool isSquare();
class Rectangle - private

class Rectangle {
    public:
        // see previous slide
        void input(const istream& in);
        void output(const ostream& out);

    private:
        int length;  // member variables
        int height;
        bool check_dimensions();
};