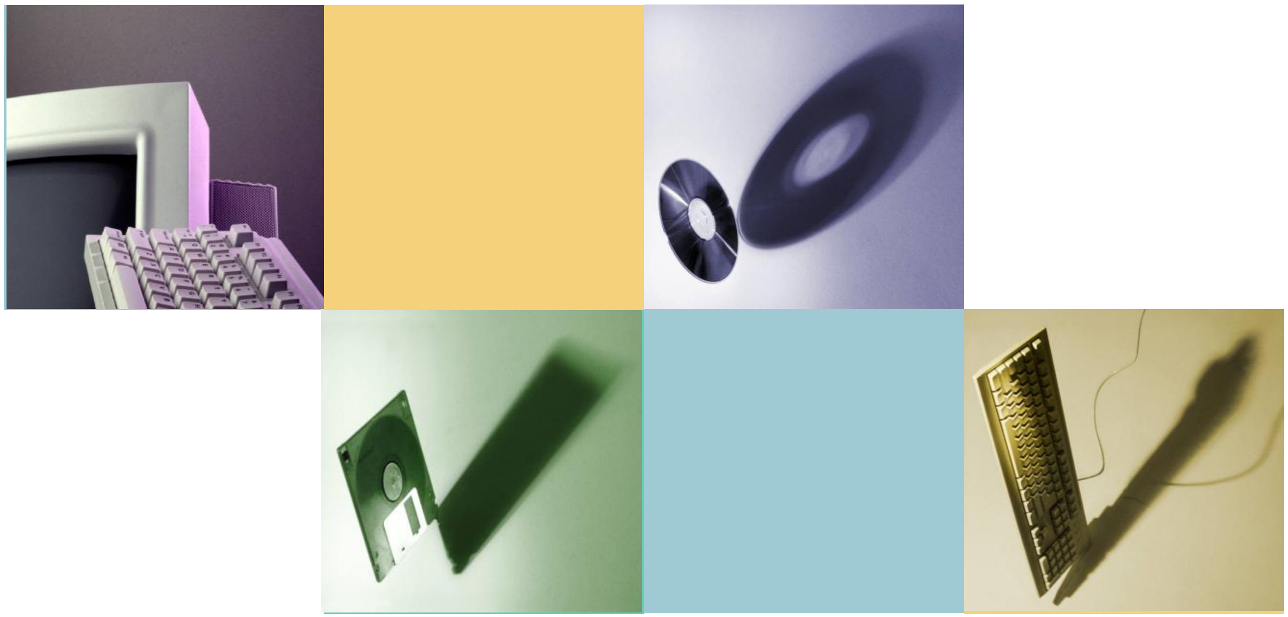
Object-Oriented Programming



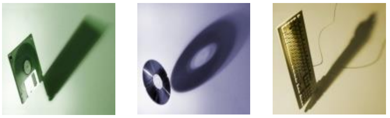
Chuan-Kang Ting

Dept of Computer Science and Information Engineering
National Chung Cheng University

Chapter 6

Structures and Classes

Now we’re going to the world of **objects**…



Outline

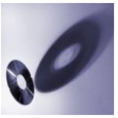
- Introduction
- Structures
- Classes

Outline

- Introduction
- Structures
- Classes

Introduction

- **Classes**
  - Perhaps the single most significant feature separating C++ from C
  - A **class** is a type whose values are called **objects**
- **Objects**
  - Have both
    - member data
    - member functions
  - These objects are the objects of OOP

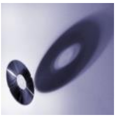




Introduction (cont'd)

- **Structure**
  - An object without any member function (really?!)
  - A collection of data items of diverse types
    - Array: a collection of variables of same type

Outline

- Introduction
- **Structures**
- Classes


Structures (1)

- **Structure types**
  - Collection of values of different types

```
struct Point
{
   double x;
   double y;
   long color;
};
```

structure tag
(usually spelled starting with an uppercase letter)

member names

structure definition
usually placed outside any function definition, i.e. **globally**

Structures (2)

- **Structure types** (cont'd)
  - Once a structure type has been given, the structure type can be used **just like the predefined types**, such as `int`, `char`, and so on

structure type Point

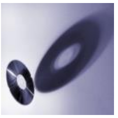
```
Point pickedPoint;

pickedPoint.x = 10.6;
pickedPoint.y = 218.7;
pickedPoint.color = 0xFF00FF;
```

member values

structure value
a collection of member values

dot

member variables
x, *y* are of type `double`
color is of type `long`


Structures (3)

- **Structure value can be viewed as**
  - A collection of member values
  - A single (complex) value
    - Therefore, use
      - structure types as predefined types
      - structure variables as variables of predefined types
      - structure values as values of variables

Structure (4)

- **Structure assignments**
  - If we declare

```
Point firstPoint, secondPoint;
```

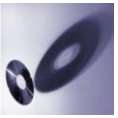
- Both are variables of "structure type **`Point`**"
- You can assign structure values by

```
firstPoint = secondPoint;
```

equivalent to

```
firstPoint.x = secondPoint.x;
firstPoint.y = secondPoint.y;
firstPoint.color = secondPoint.color;
```

→Just use structure variables in the same ways that you use simple variables of the predefined types such as `int`


Structures as Function Arguments

- **A structure type supports**
  - Call-by-value parameter
  - Call-by-reference parameter
  - The value returned by a function

```
Point generateNeighbor(Point currentPoint)
{
  Point temp;
  temp = currentPoint;
  temp.x += (5.0 - rand() % 10);
  temp.y += (10.0 - rand() % 20);
  return temp;
}
```

Initializing Structures

- **Initializing at declaration**
  - Giving a list of the member values enclosed in { }
  - Corresponding to the order of member variables in the structure type definition
  - If the list is incomplete:
    - Initialize data members **in order**
    - Initialize data members **without** initializer to zero

```
Point testPoint1 = {100, 200, 0xFFFF00};
Point testPoint2 = {150, 100};
Point testPoint3 = {400};
Point testPoint4 = {300, 300, 100, 0xFF0000};
```

Error: #initializers > #members

Outline

- Introduction
- Structures
- **Classes**
  - Defining Classes
  - Encapsulation
  - Public and Private Members
  - Accessor and Mutator Functions
  - Interface and Implementation

Classes

Class = Structure with

member data

member functions

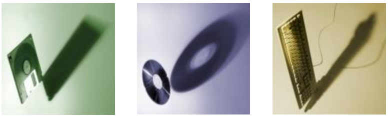
(Recall the definition of structures)

Classes (cont'd)

- **Classes/Objects**
  - The value of a variable of a class type is called an object

Loosely speaking, the **variable** of class is also often called object

- **Programming**
  - Structural? Object-oriented?
    - Purpose
    - Philosophy
      - When programming with classes, a program is viewed as *a collection of interacting objects*
  - (UML class diagram)



Class Definition (1)

- **Defining classes**
  - Similar to definition of a structure, but adding member functions
  - Normally, it contains only the declaration for its member function
    - The **definitions** for the member functions are usually given elsewhere
  - e.g.

access specifier

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

member function declaration

Class Definition (2)

- **Declaring objects** (← not classes)
  - In the same way as variables of the *predefined types* (`int`, `char`, …) and *structure types*

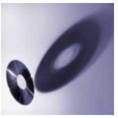
```
DayOfYear today, birthday;
```

  - The dot operator
    - Specifies a **member variable**

```
cout << today.month;
```

    - Invokes a **member function**

```
today.output();
```


Class Definition (3)

- **Defining member function**
  - To tell what a member function is a member of
    - The *scope resolution operator* **::** is used with a **class** name
    - The *dot operator* **.** is used with **objects** (i.e. with class variables)

type qualifier (class name)

member function name

```
void DayOfYear::output()
{
  switch(month)
  {
    case 1:
      cout << "Jan"; break;
    ...
  }
}
```

xxx.month?

member function definition

Class Definition (4)

- **Defining member function** (cont'd)
  - The definition of output will apply to **all objects** of type `DayOfYear`
  - At definition, we don't know the name of the objects that we will use → We CANNOT give their names at this time
  - All the member names in the function definition are specialized to the calling object

```
today.output();
```

```
{
  switch(today.month)
  {
    case 1:
    ...
  }
}
```

Class Definition (5)

- **In the definition of a member function**
  - All the member names are specialized to the **calling object**
  - You can directly use the names of all members (including variables and functions) of that class without using the **dot** operator

In a Nutshell

- **Defining classes**
  - Declaring member variables (data)
  - Declaring member functions (operators)
- **Declaring objects**
  - An object is a variable of a class type
- **Defining member functions**
  - Normally, member functions are defined outside the definition of classes

Encapsulation (1)

- **A data type consists of**
  - Data (values)
  - Operations (on these values)

e.g. `int` data type

- data (integer)
- operators +, -, *, /,...

Encapsulation (2)

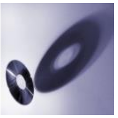
- **Abstract data type (ADT)**
  - Even you use the type, you do NOT have access to the details of how the values and operators are implemented
  - Abstract → Programmer don’t know the details
    - c.f. procedural abstraction
- **Predefined types are ADTs**
  - e.g. `int` (you don’t know how + and * are implemented)
- **Programmer-defined type**
  - Classes should also be ADTs
  - The details of *how the “operations” are implemented* should be **hidden** from any programmer who uses the class

Encapsulation (3)

- **Encapsulation**
  - Defining a class so that the implementation of the member functions and the data in objects are **not known**, or is at least **irrelevant**, to the programmer who uses the class
  - Also called information hiding and data abstraction
    - cf. procedural abstraction
  - This principle is one the main tenets of OOP (recall: **Pie**)
  - How?

    → Make all member variables private

Think about:
Structures vs. **Classes**


Public vs. Private

- **Private:**
  - The following items can only be referenced by name within the definitions of the member functions of this class

  → The value of a private member variable can only be changed by the member functions of the class

- **Public:**
  - The following items can be referenced by name anyplace

  → No restrictions on the use of public members

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

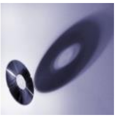
Public vs. Private (cont'd)

- **Good programming practices**
  - All member variables are *private*
  - Most member functions are *public*
- **Styling**
  - Typically, public members first
  - Of a class, the members without public and private specifier will automatically be private

Accessor and Mutator

- **General rule**
  - You should always make *all* member variables in a class private
  - However, sooner or later we need to do something with the data → accessor and mutator functions
- **Accessor functions**
  - Allow you to read the data
  - getXxx, getYyy
- **Mutator functions**
  - Allow you to change the data
  - setXxx, setYyy

```cpp
class DayOfYear
{
  public:
     void output();
  private:
     int month;
     int day;
     bool isBissextile;
};
```


Accessor and Mutator (cont'd)

- **I/O functions**
  - I/O, e.g. `input()` and `output()`, are usually just called I/O functions, even though they are mutator and accessor functions

Example (1)

Display 6.4 **Class with Private Members** (part 1 of 3)

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

class DayOfYear
{
public:
    void input( );
    void output( );
    void set(int newMonth, int newDay);
    //Precondition: newMonth and newDay form a possible date.

    void set(int newMonth);
    //Precondition: 1 <= newMonth <= 12
    //Postcondition: The date is set to the first day of the given month.

    int getMonthNumber( ); //Returns 1 for January, 2 for February, etc.
    int getDay( );
private:
    int month;
    int day;
};

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

This is an improved version of the class **DayOfYear** *that we gave in Display 6.3.*

Mutator functions

Accessor functions

Private members

Example (2)

```
    bachBirthday.set(3, 21);
    cout << "J. S. Bach's birthday is ";
    bachBirthday.output( );
    cout << endl;
    if ( today.getMonthNumber( ) == bachBirthday.getMonthNumber( ) &&
                today.getDay( ) == bachBirthday.getDay( ) )
        cout << "Happy Birthday Johann Sebastian!\n";
    else
        cout << "Happy Unbirthday Johann Sebastian!\n";

    return 0;
}

//Uses iostream and cstdlib:
void DayOfYear::set(int newMonth, int newDay)
{
    if ((newMonth >= 1) && (newMonth <= 12))
        month = newMonth;
    else
    {
        cout << "Illegal month value! Program aborted.\n";
        exit(1);
    }
    if ((newDay >= 1) && (newDay <= 31))
        day = newDay;
    else
    {
        cout << "Illegal day value! Program aborted.\n";
        exit(1);
    }
}
```

Note that the function name **set** *is overloaded. You can overload a member function just like you can overload any other function.*

Mutator functions

Example (3)

```
//Uses iostream and cstdlib:
void DayOfYear::set(int newMonth)
{
    if ((newMonth >= 1) && (newMonth <= 12))
        month = newMonth;
    else
    {
        cout << "Illegal month value! Program aborted.\n";
        exit(1);
    }
    day = 1;
}

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

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

//Uses iostream and cstdlib:
void DayOfYear::input( )
{
    cout << "Enter the month as a number: ";
    cin >> month;
    cout << "Enter the day of the month: ";
    cin >> day;
```

Accessor functions

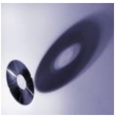
Private members may be used in member function definitions (but not elsewhere).

Interface vs. Implementation

- **Interface** **(or API)**
  - The rule for **how to use** the class
  - For a C++ class, the interface consists of
    - Comments

      → how to use the class
    - Public member functions with comments for them

      → how to use the member functions
- **Implementation**
  - Tells **how** the class interface **is realized** as C++ code
  - The implementation of a C++ class consists of
    - Private members
    - Definition of public/private member functions


Interface vs. Implementation (cont'd)

- **A well-designed class**
  - Users need **only** know the *interface* for the class
  - Users need **not** know the details of the *implementation* of the class

→ A class whose interface and implementation are separated in the above way is called an ADT or a nicely encapsulated class

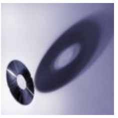
  - **Benefits**?

Structures vs. Classes

- **Structures**
  - Are normally used with
    - **all** member variables *public*
    - **no** member functions
  - However, a C++ structure can do anything a class can do, including public/private member variables and member functions
    - Most programmers don't use structures in this way, in order to distinguish the concepts of structures and classes
  - No specifier in the first group of members
    - A structure assumes the group is public
    - A class assumes the group is private

Well-designed Classes:

- All member variable **private**
- Most member function **public**

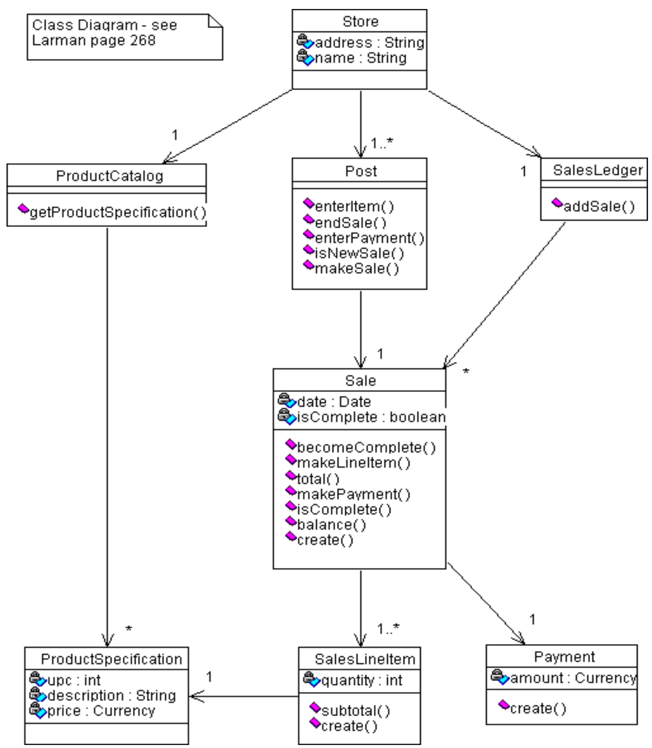

Thinking Objects

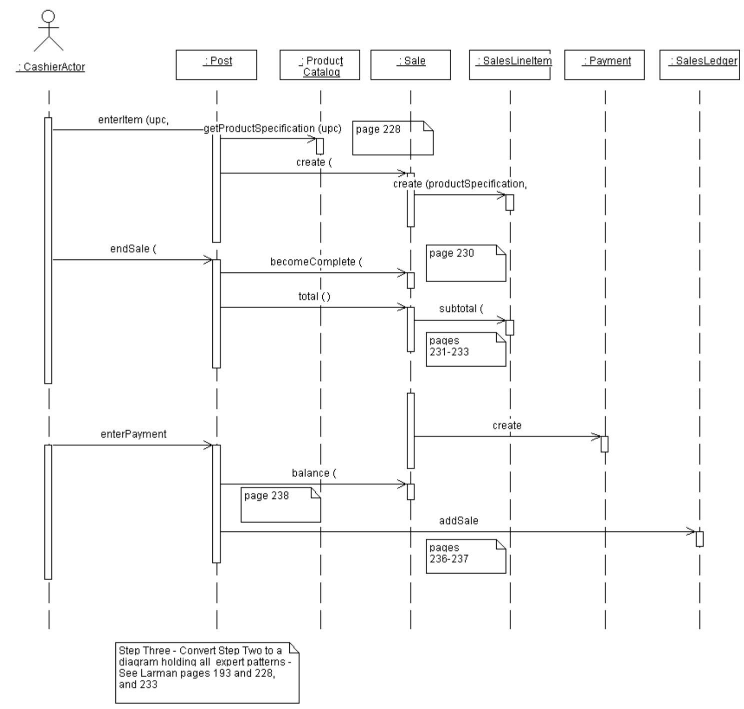
- **“Object”-oriented programming**
  - Data-centric, rather than algorithm-centric
    - Data-centric: The algorithms are developed to fit the data
    - Algorithm-centric: Design the data to fit the algorithms
  - Best style(!)
    - No global functions at all
    - Only classes with member functions

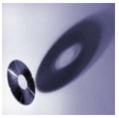
    → Programming becomes a job of defining objects and how the objects interact, rather than algorithms that operate on data

Thinking Objects (cont'd)

- **OOP**
  - Defining objects and how the objects interact, rather than algorithms that operate on data





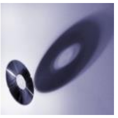


Summary (1)

- **Structures**
  - A collection of data items of diverse types
  - Structure type: Use it as predefined type
    - Call-by-value parameter
    - Call-by-reference parameter
    - Returned value
  - Assignments
  - Initialization

Summary (2)

- **Classes**
  - A class is basically a structure with member functions and member data
  - An object is (a value of) a variable of a class type
  - Class definition
    - Declaring member variables and member functions
    - Declaring objects
    - Defining member functions
  - Encapsulation
    - Defining a class so that *the implementation of the member functions and the data in objects* are not known, or is at least irrelevant, to the programmer who uses the class


Summary (3)

- **Classes** (cont'd)
  - Public vs. private
  - Accessor and mutator functions
  - Separating interface and implementation
  - Thinking objects – *real* ***object****-oriented programming*