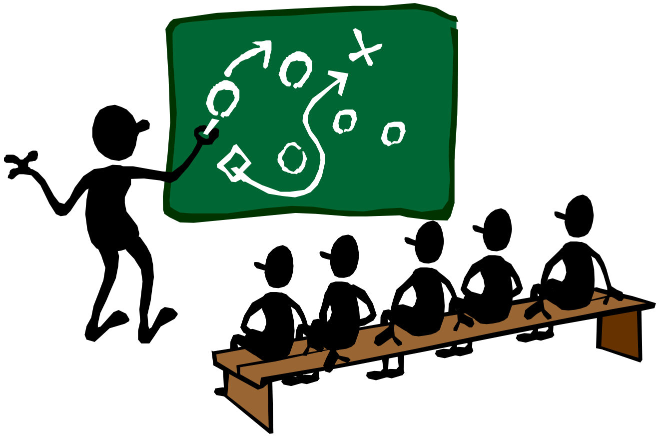
C++ Programming Language
Chapter 6 Structures and Classes



Juinn-Dar Huang
Associate Professor
jdhuang@mail.nctu.edu.tw

February 2011

Learning Objectives

- Structures
  - structure types
  - structures as function arguments and parameters
  - initializing structures
- Classes
  - definition and member functions
  - public and private members
  - notion of abstraction and encapsulation
  - principles of object-oriented programming (OOP)
  - structures vs. classes

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Structures

- Also an aggregate data type ➔ struct
- Recall: aggregate means grouping
  - array: collection of elements of same type
  - structure: collection of elements of different types
- Treated as a single item, like array
- Major difference: must define struct first!
  - yes, you are defining your own type!
  - user-defined type (vs. built-in type: int, double, …)
  - struct definition prior to any object declaration/definition

Structure Types

- Typically, struct is defined outside any functions
  - i.e., global name scope
- No memory is actually allocated
  - just to specify what the content exactly is for an object of this type
- Definition:

```
struct CDAccountV1          // name of structure type
{
    double balance;         // data member declaration
    double interestRate;
    int term;
};
```

- Structure type is user-defined type

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Structure Variables

- After a structure type is defined, you can define/declare variables of this new type

```
CDAccountV1 account;
```

- Just like defining variables of built-in types
  - i.e., int, double, …
- Variable account is of type CDAccountV1
  - each data member of account has its own value

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Accessing Structure Members

- Use dot (.) operator to access members

```
account.balance          // behaves as a double
account.interestRate     // behaves as a double
account.term             // behaves as an int
```

- Data members
  - “parts” of a structure variable
  - different structs can have data members of same name

```
struct S1 {
    int m;
    double n;
};

struct S2 {
    int n;
    float m;
};
```

Structure Example (1/3)

Display 6.1 A Structure Definition

```
//Program to demonstrate the CDAccountV1 structure type.
#include <iostream>
using namespace std;

//Structure for a bank certificate of deposit:
struct CDAccountV1
{
    double balance;
    double interestRate;
    int term;//months until maturity
};


void getData(CDAccountV1& theAccount);
//Postcondition: theAccount.balance, theAccount.interestRate, and
//theAccount.term have been given values that the user entered at the keyboar
```

An improved version of this structure will be given later in this chapter.

not a good coding style

Structure Example (2/3)

```
int main( )
{
    CDAccountV1 account;
    getData(account);

    double rateFraction, interest;
    rateFraction = account.interestRate/100.0;
    interest = account.balance*(rateFraction*(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;
}
```

(continued)

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Structures and Classes

Structure Example (3/3)

Display 6.1 A Structure Definition

```
//Uses iostream:
void getData(CDAccountV1& theAccount)
{
    cout << "Enter account balance: $";
    cin >> theAccount.balance;
    cout << "Enter account interest rate: ";
    cin >> theAccount.interestRate;
    cout << "Enter the number of months until maturity: ";
    cin >> theAccount.term;
}
```

Sample Dialogue

Enter account balance: **$100.00**
Enter account interest rate: **10.0**
Enter the number of months until maturity: **6**
When your CD matures in 6 months,
it will have a balance of $105.00

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Structure Pitfall

- Semicolon after structure definition is a must
  - trailing **;** must exist: (yes, it is rare in C++ indeed)

```
struct WeatherData
{
  double temperature;
  double windVelocity;
}; // 🡰 REQUIRED semicolon!
```

  - you **CAN** declare variables of this structure type before **;**

```
struct S1 {
    int ival;
    double dval;
} a_var, b_var ;            // avoid doing this though it is legal
```

Structure Assignments

```
struct S1 {
        int ival;
        double dval;
};
void f() {
    S1 a, b;   // both a and b are variables of S1
    a.ival = 10;
    a.dval = 2.0;
    b = a;       // ok, default assignment operator is provided by compiler
}
```

- Assigning a struct variable to another of same type is fine
  - compiler will AUTOMATICALLY generate an assignment operator
  - its default behavior is to do **memberwise assignment**

    ➔ b.ival = a.ival; b.dval = a.dval;

  Q: b += a; // is it ok? ➔ A: Error! Discuss in later chapters

Structures as Function Arguments

- Passed just like built-in data type
  - call-by-value (also call-by-pointer-value)
  - call-by-reference

```
void func(S1 p1, S1& p2, S1 *p3, const S1& p4, const S1 *p5);

void f() {
    S1 a1, a2, a3, a4, a5;
    // …
    func(a1, a2, &a3, a4, &a5);
}
```

- Avoid passing a BIG structure using call-by-value
  - already discussed in Chapter 4, remember?

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Structures as Return Types

- Can also be returned by function
  - return type is structure type
  - return statement in function definition sends structure variable back to caller

```
S1 func(S1 p) {
    p.ival += 1;  p.dval += 1.0;
    return p;
}
void f() {
    S1 a, b;
    a.ival = 1; a.dval = 1.0;
    b = func(a);         // as a result, b.ival = 2, b.dval = 2.0;
}
```

- Similarly, avoid returning a BIG structure

Initializing Structure Variables (1/2)

- Structure variables can be initialized at definition
- Old C-style initialization method still works
  - similar to array initialization

```
struct Date {
     int month;
     int day;
     int year;
};
void () {
   Date d1 = {3, 4, 2011};     // month = 3, day = 4, year = 2011
   Date d2 = {3, 4};           // month = 3, day = 4, year = 0
   Date d3 = {3, 4, 5, 6};     // error, too many initializers
   Date d4;                    // ok, month = ?, day = ?, year = ?
   d4 = {3, 4, 2001};          // error, it's an assignment, not initialization
}
```

Initializing Structure Variables (2/2)

- Avoid initializing structure variables using old C style
  - it is just for backward compatibility with C

- A better C++ style will be introduced in Chapter 7
  - you are now learning C++ after all !

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Classes

- C++ supports object-oriented programming (OOP)
- Classes are foundation for OOP in C++

- Aim of C++ class is to provide programmer a means for creating new types that can be used as conveniently as built-in types

- C++ class is much more powerful than structure in C
  - not just data members
  - add member functions
  - add access control mechanism
  - and many more others … (in later chapters)

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Class Definitions

- Defined similar to structures
- Example:

```
class DayOfYear {                    // name of class type
public:                              // access specifier
        int month;                   // data member declaration
        int day;                     // data member declaration
        void output();               // member function declaration
};
```

- Typically, only member function's declaration in class definition
  - function's implementation is elsewhere
- Define a class ➔ Define a **new type** !

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Creating Class Objects

- Defined/declared same as variables of built-in types and structure types
- Example:

DayOfYear today, birthday;

  - define two objects of class type DayOfYear

- An object of a class owns:
  - data members it can access
    - e.g., month, day
  - member functions (operations) it can call
    - e.g., output()

Member Access

- Same way to access data members as structures
  - using dot operator (.)
  - example:

    today.month   // access today's month
    birthday.month   // access birthday's month

- Invoke member function
  - still using dot operator (.)
  - example

    today.output(); // today calls member function output()

Class Member Functions (1/3)

- Must define (i.e., implement) class member functions
- Like other function definitions
  - outside any other functions
  - must specify the class it belongs to

    void DayOfYear::output() { // …}

  - :: is called scope resolution operator
  - tell compiler what class this member function belongs to
    - different classes can have member functions with same name
  - item before :: called type qualifier
    - class name serves as type qualifier

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Class Member Functions (2/3)

```
class S1 {
    // …
    void func();   // S1's func()
};

class S2 {
    // …
    void func();   // S2's func(), ok, name can be same
};


void S1::func()   // This is S1's func()
{ // … }

void S2::func()   // This is S2's func()
{ // … }
```

Without type qualifiers, you can't tell which is which

Class Member Functions (3/3)

Notice the definition of output() (shown next slide)

- Access data members of same class directly
  - no need to define again
- Member functions can be called for all objects of that class
  - while accessing data members, referring to those of the object calling it
  - example:

    today.output();   // use today's month & day

    birthday.output();   // use birthday's month & day

Class Example (1/4)

Display 6.3 Class with a Member Function

```
//Program to demonstrate a very simple example of a class.
//A better version of the class DayOfYear will be given in Display 6.4.
#include <iostream>
using namespace std;

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

int main( )
{
    DayOfYear today, birthday;
    cout << "Enter today's date:\n";
    cout << "Enter month as a number: ";
    cin >> today.month;
    cout << "Enter the day of the month: ";
    cin >> today.day;
    cout << "Enter your birthday:\n";
    cout << "Enter month as a number: ";
    cin >> birthday.month;
    cout << "Enter the day of the month: ";
    cin >> birthday.day;
```

Normally, member variables are **private** *and not* **public***, as in this example. This is discussed a bit later in this chapter.*

Member function declaration

(continued)

→ P35

Class Example (2/4)

Display 6.3 **Class with a Member Function**

```
    cout << "Today's date is ";
    today.output( );
    cout << endl;
    cout << "Your birthday is ";
    birthday.output( );
    cout << endl;

    if (today.month == birthday.month && today.day == birthday.day)
        cout << "Happy Birthday!\n";
    else
        cout << "Happy Unbirthday!\n";
    return 0;
}
//Uses iostream:
void DayOfYear::output( )
{
    switch (month)
    {
        case 1:
            cout << "January "; break;
        case 2:
            cout << "February "; break;
        case 3:
            cout << "March "; break;
        case 4:
            cout << "April "; break;
```

Calls to the member function **output**

use data member directly !

Member function definition

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Class Example (3/4)

```cpp
        case 5:
            cout << "May "; break;
        case 6:
            cout << "June "; break;
        case 7:
            cout << "July "; break;
        case 8:
            cout << "August "; break;
        case 9:
            cout << "September "; break;
        case 10:
            cout << "October "; break;
        case 11:
            cout << "November "; break;
        case 12:
            cout << "December "; break;
        default:
            cout << "Error in DayOfYear::output. Contact software vendor.";
    }

    cout << day;
}
```

use data member directly !

Class Example (4/4)

Display 6.3 **Class with a Member Function**

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 October 15
Your birthday is February 21
Happy Unbirthday!

Dot and Scope Resolution Operators

Used to specify "of what thing" they are members

- Dot operator (.)
  - specifies a member of a particular object
  - e.g., today.month // today's month
- Scope resolution operator (::)
  - specifies what class a member belongs to
  - e.g.,

```
void DayOfYear::output()     // DayOfYear's output()
{ // …}
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

A Class Is a Type

- Classes are types defined by programmer !
- You can create variables of a class type
  - or call them objects
- Functions can have parameters of a class type
  - both call-by-value and call-by-reference
- A class type can be a return type of function
- You can use class types just like any built-in types!

➔ **All in all, a class is a type!**

Data Types

- Definition of a data type includes
  - a collection of legal values
  - a set of well-defined operations that act on those values
- Example:

  C++ built-in int : (assuming 32-bit long)
  values: $[-2^{31} \sim 2^{31}-1]$
  operations: +, –, *, /, %, <<, >>, …
- Define a new type using C++ class
  - define a collection of values ➔ data members
  - define a set of operations ➔ member functions

Abstract Data Types (ADT)

- Abstract Data Type (ADT)
  - is a data type
  - the *specification* of the objects is separated from the *representation* of the objects
  - the *specification* of the operations is separated from the *implementation* of the operations

- Again, you can implement ADT in C++ with class

Key Idea Here

Separate implementation from specification →

A specification can be achieved by different implementations

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Abstraction and Encapsulation (1/2)

- For a DVD player with functions ⏮ ⏪ ■ ▶ ⏸ ⏩ ⏭
  - do you know how above buttons work?
  - do you know what is inside a DVD player?

  ➔ abstraction

- User manual of a DVD player
  - does it tell you *how* the player *implements* ▶ ? or
  - does it tell you *what* is the result after pressing ▶ ?

  ➔ encapsulation

Abstraction and Encapsulation (2/2)

- Abstraction
  - separate the specification of an object from the implementation
  - user does not know the details of how an object is represented
- Encapsulation or information hiding
  - hide the implementation details of operations
  - user does not know the details of how operations work

Fundamental Principles of OOP

But, how to achieve in C++ class?

Member Access Control (1/2)

- Modify DayOfYear's definition in previous example

```
class DayOfYear {
 public:                // public interface to the outside world
      void input();
      void output();
private:                // private to this class only
      int month;
      int day;
};
```

- In this case, two data members are now private

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Member Access Control (2/2)

- A class can have multiple private/public sections
  - avoid doing so though it is ok to do so

```
class S1 {
public:
        // …
private:
        // …
public:
        // …
};
```

- Typically, public goes first
  - allow easy viewing of public portions for class users
  - private data is hidden, so irrelevant to users

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Private vs. Public

- Both data members and member functions can be either private or public
- Data members are usually private
  - you do not know exact representation ➔ abstraction
  - object is manipulated through member functions
- Member functions are usually public
  - you can use public interface for object manipulations
  - you still do not know how these member functions get implemented ➔ encapsulation

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Example: Public vs. Private (1/2)

- Accessing private members outside class definition is **NOT** allowed

DayOfYear today; // use definition in P32

- In Page 22,

```
cin >> today.month;    // error, NOT ALLOWED!
cin >> today.day;      // error, NOT ALLOWED!
```

  - must set month and day through a new public member function input()

➔ P22

Example: Public vs. Private (2/2)

```
int main( ) {
    DayOfYear today, bachBirthday;  // use definition in P32
    cout << "Enter today's date:\n";
    today.input( );   // ok, call public member function
    cout << "Today's date is ";
    today.output( ); // ok, call public member function
    cout << endl;

     return 0;
}

void DayOfYear::input( ) {
    cout << "Enter the month as a number: ";
    cin >> month;   // ok to access private data member
    cout << "Enter the day of the month: ";
    cin >> day;       // ok to access private data member

     if ((month < 1) || (month > 12) || (day < 1) || (day > 31)) {
       cerr << "Illegal date! Program aborted.\n";
       exit(1);
    }
}
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Structures and Classes

Accessors and Mutators

- Two types of member functions
  - accessor
  - mutator
- Accessors
  - read data members ONLY
  - NEVER modify data members in any circumstances
- Mutators
  - can modify data members

Q: Why is the type of member function important?

A: You will see in the next Chapter

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Separate Interface from Implementation

- You need not know the details of how a class is implemented
  - principles of OOP ➔ abstraction and encapsulation
- You only needs to know how to manipulate class objects through public interface
  - in C++ ➔ public members (mostly, functions)
- Implementation details of class are hidden
  - member function definitions are elsewhere
  - you do not have to know at all!
    (sometimes you simply cannot see them even if you want)

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Thinking Objects

Programming paradigm shift

- In the past
  - handle data structures and functions separately
- OOP era
  - define class by considering both data structures and functions at the same time
  - handle interactions among objects
- Designing software solutions
  - define variety of class objects and how they interact

Benefits from OOP (1/2)

- A class can be implemented in different ways as long as the public interface is unchanged
  - changes in the internal implementation is OK since implementation is not visible outside
  - the best part is: in fact, you WON'T even notice changes since you can only manipulate class objects through the public interface
- Implementation changes of one class won't cause you to rewrite the rest part of program!

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Benefits from OOP (2/2)

- Easier debugging and testing
  - due to encapsulation
- Better reusability
  - due to abstraction and encapsulation
  - Standard Template Library (STL)
    - discuss in Chapter 19

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Structures vs. Classes (1/2)

- In fact (it's not a joke, I promise you), in C++
  - a structure is a class
  - a class is a structure
  - a structure assumes all members public by default
  - a class assumes all members private by default

```
struct S1 {
an implicit public:
      // public members
};
```

```
class C1 {
an implicit private:
      // private members
};
```

```
struct C1 {
private:
      // …
};
```

Equivalent

```
class C1 {

      // …
};
```

Structures vs. Classes (2/2)

- Technically, class and struct are almost identical!
  - Yep! struct can have member functions too
  - Yep! struct can have private members too
- Why does C++ still provide structures?
  - provides backward compatibility with C
- In practice
  - a struct should make all data members public
  - a struct should not have member functions
  - that is, still regard struct the same way in C
    - avoid confusion!

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Summary (1/2)

- Structure is a collection of elements of different types
- Class combines data and functions into a single entity
- Structures and classes are types
- Class and structure types can be argument/parameter/return types of function
- Member access control
  - public: can be accessed outside class
  - private: can be accessed only in member functions' definitions
- In C++ class
  - data members are typically (but not necessarily) private
  - member functions are typically (but not necessarily) public

Summary (2/2)

- Principles of OOP
  - abstraction and encapsulation
  - separate specification from implementation
- C++ support OOP through classes
- A C++ class design
  - consists of two key parts
    - interface: what user needs for proper object manipulation
    - Implementation: details of how an entire class works
- In C++, still use structures the same way in C

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw