Classes and Structures

Computer Programming for Engineers (DSAF003-42)

Instructor:

Youngjoong Ko (nlp.skku.edu)

Recap

- **String**
  - C-string ends with a null character.
  - `std::string` for C++
  - Lots of member functions for string class were introduced.
- **Console Input/Output**
  - `cin` and `cout`
  - `getline` for receiving a whole line
    - A weird behavior were demonstrated with a mixed usage of `cin` and `getline`.

Today

- **Structures**
  - Structure types
  - Structures as function arguments
  - Initializing structures
- **Classes**
  - Defining, member functions
  - Public and private members
  - Accessor and mutator functions
  - Structures vs. classes

STRUCTURES

Structures

- **2nd aggregate data type: struct**
  - aggregate meaning "grouping"
  - Structure: heterogeneous collection of values of different types
  - Recall array: homogeneous collection of values of same type

- **Treated as a new data type**
  - Major difference: Must first "define" `struct` prior to declaring any variables

Structure Types

- **Define `struct` globally (typically)**
  - No memory is allocated
  - Just a "placeholder" for what our `struct` will "look like"

- **Definition:**

```
struct CDAccountV1  // name of new struct "type"
{
   double    balance; // member names
   double    interestRate;
   int       term;
};
```

Declaring/Instancing Structure Variable

- **Now we can declare variables of this new type:**

```
CDAccountV1 account;
```

  - This is creating an instance of the structure.
  - Just like declaring simple types
  - Variable account now of type CDAccountV1

Note: In C++, the struct keyword is optional before in declaration of a variable. In C, it is mandatory.

Accessing Structure Members

- **Dot(.) Operator to access members**

```
account.balance;
account.interestRate;
account.term;
```

- **Called "member variables"**
  - The "parts" of the structure variable
  - Different `structs` can have same name member variables

Structure Pitfalls

- **Semicolon after structure definition**
  - Semicolon (;) MUST exist in the end of the declaration:

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

  - Required since you "can" declare structure variables in this location

Structures as Function Arguments

- **Passed like any simple data types**
  - Pass-by-value
  - Pass-by-reference
    - Recommended, when the size of a structure is large
    - Avoids the redundant copy of the data
  - Or combination

- **Can also be returned by a function**
  - The return-type is a structure type.
  - The return statement in the function definition sends a structure variable back to the caller.

structure member names 1


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

struct CDAccountV1 { // name of new struct "type"
  double  balance; // member names
  double  interestRate;
  int     term;
};

struct CDAccountV2 { // name of new struct "type"
  int   balance; // member names
  int   interestRate;
  int   term;
} account2;

int main()
{
  // C++
  CDAccountV1 account1;
  // C
  //struct CDAccountV1 account1;
```

structure member names 2


```
  account1.balance = 1000;
  account1.interestRate = 0.02;
  account1.term = 2;

  cout << "I have $" << account1.balance << " in my account." << endl;
  double rate1 = pow(1+account1.interestRate, account1.term);
  cout << "After " << account1.term << " years it will become $" << account1.balance * rate1 << "." << endl;

  // We can use the same names for member vars of different structs
  account2.balance = 2000;
  account2.interestRate = 0.02; // CHECK TYPE!
  account2.term = 5;

  cout << "I have $" << account2.balance << " in my account." << endl;
  double rate2 = pow(1+account2.interestRate, account2.term);
  cout << "After " << account2.term << " years it will become $" << account2.balance * rate2 << "." << endl;

  return 0;
}
```

Passing struct as argument


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

struct CDAccountV1 { // name of new struct "type"
  double  balance; // member names
  double  interestRate;
  int     term;
};

void printAccountInfo(CDAccountV1 myAccount) {
  cout << "I have $" << myAccount.balance << " in my account." << endl;
  double rate = pow(1+myAccount.interestRate, myAccount.term);
  cout << "After " << myAccount.term << " years it will become $" << myAccount.balance * rate << "." << endl;
  // What happens when we modify the value of myAccount's member variables?
}

int main() {
  CDAccountV1 acc;
  acc.balance = 2000;
  acc.interestRate = 0.02;
  acc.term = 3;
  printAccountInfo(acc);
  return 0;
}
```

Initializing Structures

- **Aggregate initialization**
  - Declaration provides initial data to all three member variables

```
struct Date
{
    int month;
    int day;
    int year;
};
Date dueDate = {12, 31, 2003};
```

- **Non-static data member with initializer (C++11)**
  - This can be even simpler by providing default values in declaration.

```
struct Date
{
    int month = 12;
    int day = 31;
    int year = 2003;
};
```

CLASSES

Classes

- **Similar to structures**
  - Simply Adds member FUNCTIONS as well as member variables.

```
class DayOfYear // name of new class type
{
public:
  void output(); // member function! implementation elsewhere
  int  month;
  int  day;
};
```

- **Integral to object-oriented programming**
  - Focus on objects containing both data and operations.
  - We can define object's behavior using the member functions.

Declaring Objects

- **Declared same as all variables**
  - Predefined types, structure types
- **Example:**

```
DayOfYear today, birthday;
```

  - Declares two objects of class type DayOfYear
- **Objects include:**
  - Data members: month, day
  - Operations (member functions): output()

Class Member Access

- **Members accessed same as structures**

```
today.month;
today.day;
today.output(); // invokes member function
```

- **Dot (.) and Scope Resolution (::) Operator**
  - Used to specify "of what thing" they are members
  - Dot operator: specifies member of particular object
  - Scope resolution operator: specifies what class the function definition comes from

Class Member Functions Definition

- **Must define or "implement" class member functions**
- **Like other function definitions**
  - Must specify class:

```
void DayOfYear::output()
{
    …
}
```

  - :: is scope resolution operator
  - It instructs the compiler "what class" member is from.

`this` Pointer in Member Function

- **`this` pointer:**
  - **Predefined pointer** to the calling object **itself**

```
void DayOfYear::output()
{
    cout << this->month << endl;
    cout << this->day << endl;
}
```

  - `'->'`: member dereference operator (the same as in C)
- **Scope conflict**
  - When member function parameter has the same name, we can only access the data member using `this`.

```
void DayOfYear::assign ( int month, int day )
{
    this->month = month;
    this->day = day;
}
```

Complete Class Example

- **Display 6.3 Class With a Member Function (1 of 5)**

```
//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( ); //Member function declaration
    int month;
    int day;
    /*Normally, member variables are private
    and not public, as in this example. This is
    discussed a bit later in this chapter./*
};
int main( )
{
    DayOfYear today, birthday;
    cout << "Enter today's date:\n";
```

Complete Class Example

- **Display 6.3 Class With a Member Function (2 of 5)**

```
    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;
    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;
}
```

Complete Class Example

- **Display 6.3 Class With a Member Function (3 of 5)**

```
//Uses iostream:
void DayOfYear::output( ) //Member function definition
{
    switch (month)
    {
        case 1:
            cout << "January "; break;
        case 2:
            cout << "February "; break;
        case 3:
            cout << "March "; break;
        case 4:
            cout << "April "; break;
        case 5:
            cout << "May "; break;
        case 6:
            cout << "June "; break;
        case 7:
            cout << "July "; break;
```

Complete Class Example

- **Display 6.3 Class With a Member Function (4 of 5)**

```
        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.";
    }
    cout << day;
}
```

Complete Class Example

- **Display 6.3 Class With a Member Function (5 of 5)**

```
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!
```

member functions and this 1


```
#include <iostream>
using namespace std;
class DayOfYear { // name of new class type
  public:
    void output(); // member function! implementation elsewhere
    void assign1(int m, int d);
    void assign2(int month, int day);
    void assign3(int, int);
    int month;
    int day;
};

void DayOfYear::assign1(int m, int d) {
  month = m;
  day = d;
}

void DayOfYear::assign2(int month, int day) {
  this->month = month;
  this->day = day;
}
```

member functions and this 2


```
void DayOfYear::assign3(int month, int day){
  this->month = month;
  this->day = day;
}
void DayOfYear::output() {
  cout << month << "/" << day << endl;
}

int main() {
  DayOfYear birthday;

  birthday.month = 5;
  birthday.day = 11;
  birthday.output(); // invokes member function

  birthday.assign1(9,6);
  birthday.output();

  birthday.assign2(1,22);
  birthday.output();

  birthday.assign3(12,23);
  birthday.output();
}
```

CLASS DETAILS

A Class's Place

- **Class is full-fledged type!**
  - Just like data types int, double, etc.
- **Can have variables of a class type**
  - We simply call them "objects"
- **Can have parameters of a class type**
  - Pass-by-value
  - Pass-by-reference
- **Can use class type like any other type!**

Encapsulation

- **Any data type includes**
  - Data (range of data)
  - Operations (that can be performed on data)
  - Example:
    - `int` data type has:
    - Data: -2147483648 to 2147483647 (for 32 bit int)
    - Operations: +, -, *, /, %, logical, etc.

- **Same with classes**
  - But WE specify data, and the operations to be allowed on our data!

Abstract Data Types (ADTs)

- **Abstract Data Types (ADTs):**
  - Collection of data values together with set of basic operations defined for the values
    - e.g., stack: data values and behaviors (push, pop, empty, top,...)
  - **"Abstract"**: programmers don't know details
    - Also, don't have to know the details.

- **ADT's often "language-independent"**
  - We implement ADT's in C++ with classes:
  - Other languages implement ADT's as well

Principles of OOP

- **Principles of OOP (Object-Oriented Programming)**
  - **Information hiding**
    - Details of how operations work not known to "user" of class
  - **Data Abstraction**
    - Details of how data is manipulated within ADT/class not known to user
  - **Encapsulation**
    - Bring together data and operations, but keep "details" hidden

Thinking Objects

- **Focus for programming changes**
  - Before → algorithms center stage
  - OOP → data is focused
- **Algorithms still exist**
  - They simply focus on their data
  - Are "made" to "fit" the data
- **Designing software solution**
  - Define variety of objects and how they interact

Public and Private Members

- **Data in class almost always designated private in definition!**
  - Upholds principles of OOP
  - Hide data from user
  - Allow manipulation only via operations/member functions

- **Public items (usually member functions) are "user-accessible"**

Public and Private Qualifiers

- **Modify previous example:**

```
class DayOfYear
{
public: // can be directly accessed even from non-member functions
    void input();
    void output();
private: // can be directly accessed only in member functions
    int month;
    int day;
};
```

- **Data now private, while member functions are public**

```
cin >> today.month;   // NOT ALLOWED!
cout << today.day;    // NOT ALLOWED!
today.output();       // ALLOWED!
```

Public and Private Style

- **Can mix & match public & private**
- **More typically place public first**
  - Allows easy viewing of portions that can be USED by programmers using the class
  - Private data is "hidden", so irrelevant to users

public vs private


```
#include <iostream>
using namespace std;
class DayOfYear {
  public:
    void output(); // member function! implementation elsewhere
    void assign(int month, int day);
  private:
    int month;
    int day;
};
void DayOfYear::output() {
  cout << month << "/" << day << endl;
}
void DayOfYear::assign(int month, int day) {
  this->month = month;
  this->day = day;
}
int main() {
  DayOfYear birthday;
  // Illegal accesses to private member variables
  //birthday.month = 5;
  //birthday.day = 11;
  birthday.assign(5, 11);
  birthday.output(); // invokes member function
  return 0;
}
```

Accessor and Mutator Functions

- **Object needs to "do something" with its data**
- **Call accessor member functions**
  - Allow object to read data
  - Also called "get member functions"
  - Simple retrieval of member data
- **Mutator member functions**
  - Allow object to change data
  - Manipulated based on application

`const`: Usages in Member Functions

- **Accessors typically accompany const after:**

```
class DayOfYear {
public:
   int get_day() const; // typical accessor definition
};
```

- **Usages of `const`:**

```
struct DayOfYear{
   const int* const get_pointer_to_day() const;
};
```

  - first `const`: the value referenced by the pointer is constant (immutable)
  - second `const`: the pointer itself is constant
  - third `const`: modification to any class member variables are not allowed in the function.

const modifier


```
#include <iostream>
using namespace std;
int main() {
  int number7 = 7;
  int number8 = 8;
  int* const addr_const = &number7;
  const int* const_addr = &number7;
  cout << number7 << " has an address of " << addr_const << endl;
  cout << number7 << " has an address of " << const_addr << endl;

  // ERROR: To check the address of number8
  //addr_const = &number8;
  //cout << number8 << " has an address of " << addr_const << endl;
  const_addr = &number8;
  cout << number8 << " has an address of " << const_addr << endl;

  // Assigning a new value through pointers, addr_const points to number7
  *addr_const = 77;
  cout << "number7 is now " << *addr_const << endl;

  // ERROI: const_addr points to number8
  //*const_addr = 88;
  //cout << "number8 is now " << *const_addr << endl;
}
```

Separate Interface and Implementation

- **Users of class need not see details of how class is implemented**
  - Principle of OOP → encapsulation
- **Users only need "rules"**
  - Called "interface" for the class
    - In C++: public member functions and associated comments
- **Implementation of class is hidden**
  - Member function definitions elsewhere
  - Users need not see them

Structures vs. Classes

- **Technically, they are the same**
  - **Unlike C, structures can have member functions!**
  - Perceptionally different mechanisms
- **Structures**
  - Default qualifier is public (private for classes)
- **Classes**
  - Default qualifier is private
  - Typically all data members private
  - Interface member functions public