Classes

C++ Object Oriented Programming
Pei-yih Ting
93/02 NTOU CS

Contents

- Encapsulation
- Access Specifiers
- Default Access
- Private Data
- Public vs. Private Functions
- Object State
- Scope
- Inline Member Functions
- Constant Functions
- Accessor and Mutator

Adding Member Functions

- Evolving from **struct** to **class**: functions in a struct: the interface of an object

```
struct Data
{
   int m_x;
   int m_y;
   void setValues(int inputX, int inputY);
   int add();
};
void main()
{
   Data myData;
   myData.setValues(2, 3);
   cout << myData.add();
}
void Data::setValues(int inputX, int inputY)
{
   m_x = inputX;   m_y = inputY;
}
int Data::add()
{
   return m_x+m_y;
}
```

data members

defining a new type

member function declarations (prototypes)

an object

calling the member functions: sending message to the object

definitions of member functions

access data members directly

Member Functions (cont'd)

- Try calling one of the member functions without the object

```
add();
error C2065: 'add' : undeclared identifier
```

- Adding correct scope won't work either

```
Data::add();
error C2352: 'Data::add' : illegal call of non-static member function
```

- Try using one of the data members without the object

```
cout << m_x;
error C2065: 'm_x' : undeclared identifier
cout << Data::m_x;
error C2597: illegal reference to data member 'Data::m_x' in a static function
```

in main()

- Something you CAN do but you DON'T want to do

```
myData.setValues(2, 3);
myData.m_x = 4;
cout << myData.add();
```

Output:
7

Encapsulation

- How does C++ enforce the encapsulation? **Access Specifiers**

```
class Data
{
public:
   void setValues(int inputX, int inputY);
   int add();
private:
   int m_x;
   int m_y;
};
```

could use keyword **struct** instead

whatever in the public segment is the interface of a class

- What does *private* mean?
  - Private data can only be accessed in member functions
  - It does **not** mean they can only be accessed through objects
- Why does this help?

```
myData.m_x = 4;
error C2248: 'm_x' : cannot access private member declared in class 'Data'
```


Access Specifiers

- Members of a class are **private** by default, members of a struct are **public** by default

```
class Data
{
   int m_x;        <-- private
   int m_y;
public:
   void setValues(int inputX, int inputY);
   int add();
};
```

```
struct Data
{
   int m_x;        <-- public
   int m_y;
public:
   void setValues(int inputX, int inputY);
   int add();
};
```

- You can mix public and private at will, but why should you?

```
class Data
{
private:
   int m_x;
public:
   void setValues(int inputX, int inputY);
private:
   int m_y;
public:
   int add();
};
```

Style: put public segment before private segment

Data: Private? or Public?

Data members should always be private.
Member functions should be private unless they must be public.

- If data members are private, how does a client program access them?

```
myData.setValue(3, 5); myData.add();
```

- Why should a client not change the data parts directly?
  - Reason 1: Deny meddling access

```
myData.m_y = -20;  // would pass the robustness check
…
void Data::setValues(int inputX, int inputY) {
   if ((inputX == 0) || (inputY < 0)) // robustness check
      cout << "Warning: illegal data values!!";
   else
      m_x = inputX, m_y = inputY;
}
```

  - Reason 2: Change can break the client code

```
class Data { …
   char m_x;       // original client code myData.m_x = 666; would be wrong
};
```


Functions: Private? or Public?

- Why make a function public?

```
void main() {
   Data myData;
   myData.setValues(2, 3);
   cout << myData.add();
}
```

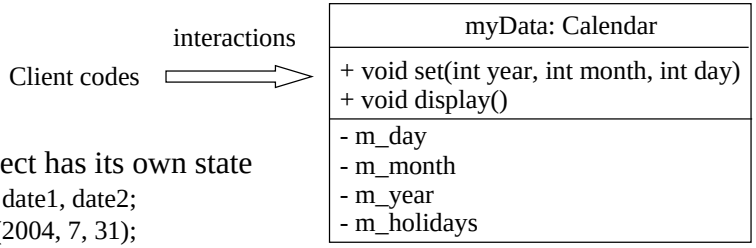
client codes demand an interface to manipulate this sort of objects, i.e. services to client codes

- Why make a function private?
  - Helper function, not a service of this class of object
  - If the programmer wants to preserve the extensibility of this piece of code
  - If the programmer cannot find any reason to make it public. (Something like "defensive driving"…call it "defensive coding")

```
class Calendar
{
…
private:
   bool isBufferEmpty();  // not a service
…
};
```

Object State

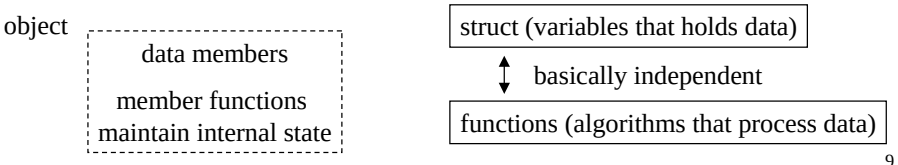
- The data members of a class comprise the state of an object



- Each object has its own state

```
Calendar date1, date2;
date1.set(2004, 7, 31);
date2.set(1970, 1, 1);
```

- Each object shares the same code for member functions
- Why calling these variables (data members) **state**?



Scope

- Two classes can have member functions or data members of the same name; member functions and data members are of class scope

```
mathObject.setValues(3, 4);
graphicsObject.setValues(4, 67);
```

```
mathObject.m_x = 10;
graphicObject.m_x = 20;
```

- Toplevel functions, variables and objects are of global scope

```
setValues(5, 6);  // or ::setValues(5, 6);     will not be ambiguous
```

- Disambiguation:

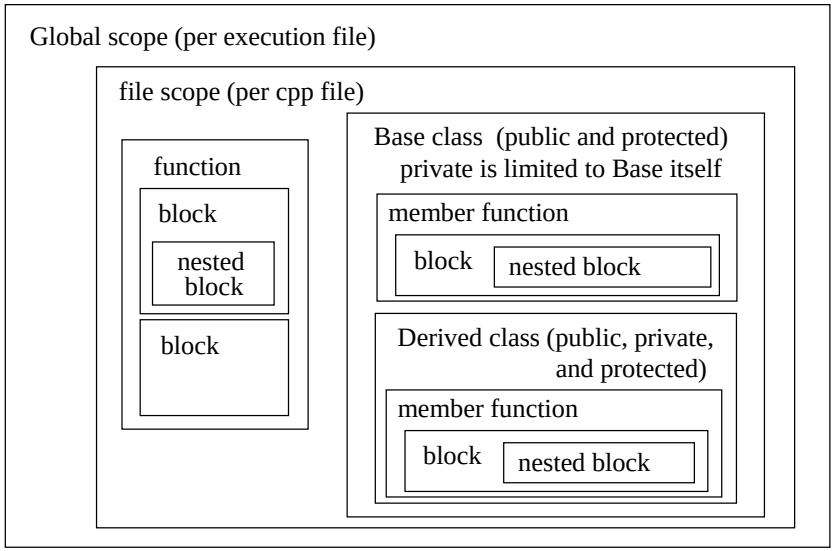
```
class Point {
… int x, y; …
};
```

```
void Point::setValues(int x, int y)
{
   Point::x = x;    // Point::x and this->y both refer to the data member of
   this->y = y;     // the class Point
}
```

- Where should classes and member functions be put into?

```
classes:                typically in the .h file
member functions:       always in the .cpp file
```


Scope (cont'd)



Inline Member Functions

- Member function can be inline

```
inline void Data::setValues(int inputX, int inputY)
{
   x = inputX;
   y = inputY;
}
```

- Inline expansion is determined by the compiler, the compiler can only expand an inline function after knowing its definition.
  - The above definition of Data::setValues() must come before any invocation
  - Another way is defining setValues() as inline in class declaration

```
class Data
{
  …
  inline void setValues(int inputX, int inputY);
  …
};
```

This way of definition is not recommended.
Reason: Don't commit the class to the inline function.

Inline Member Functions

- A function can also be defined within the class. Such a function is automatically inline.

```
class Data
{
public:
   void setValues(int inputX, int inputY)
   {
      m_x = inputX;
      m_y = inputY;
   }
   int add();
private:
   int m_x;
   int m_y;
};
```

inline

Don't define functions within the class, even though you can. This commits you to an inline function and clutters up the class definition. (JAVA's only way)

- What really happens? Inline functions are not shared by all objects of the class. Every call to the function inserts the code of the function (limited by the capability of the compiler).

Constant Functions

- A member function declared as ***const*** cannot change any data members of the class, which also means that it cannot call any other non-constant function.

```
class Data
{
public:
   void setValues(int inputX, int inputY);
   int add();      // no collision with the next one
   int add() const;
private:
   int m_x;
   int m_y;
};
int Data::add() const
{
  return m_x + m_y;
}
```

error C2662: cannot convert 'this' pointer from 'const struct Data' to 'struct Data &'. Conversion loses qualifiers.

part of the function signature

```
void main()
{
   Data obj;
   const Data *ptr=&obj;
   obj.add();       // call int add();
   ptr->add();      // call int add() const;
}
```

If there is no int Data::add(); This line will call add() const;

Accessor and Mutator

- Accessor functions: a function that returns a data member.
  - All accessor functions should be const.
- Mutator function: a function that alters object's state.
- Simple accessor and mutator functions are often inline

```
inline void Data::setX(int inputX) {
   m_x = inputX;
}
…
void main() {
   …
   object.setX(10);  // is equivalent to m_x = 10;
   …
}
```

- Simple accessor and mutator functions often mean that the design is not encapsulated well. Object boundary is not placed well. An object providing services is often abstracted better and encapsulated better.

Accessor and Mutator

- Should you provide an accessor function for every data member?
  - No, some data is internal to the class.
  - Never give the client more than is absolutely necessary.
- Should you provide a mutator function for every data member?
  - No, not necessarily.
- Ex.

```
calendarObject.setDay(14);
calendarObject.setMonth(2);
calendarObject.setYear(2004);
```

⇨ calendarObject.setDate(14, 2, 2004);

better, concise and convenient interface

```
day = calendarObject.getDay();
month = calendarObject.getMonth();
year = calendarObject.getYear();
cout << year << '/' << month << '/' << year;
```

You cannot check mutual consistency with separate mutator functions.

⇨ calendarObject.printDate();

It's a better abstraction for an object to provide a service than just be a storage.