More Classes

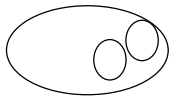
C++ Object Oriented Programming
Pei-yih Ting
NTOU CS

Contents

- Object composition and constructors
- Initialization of object within object
- Returning pointers
- ***this*** pointer
- Exploiting implicit references
- Class conversion
- Static data members
- Static member functions

Object Component

- Sometimes you would like to use a well designed object as a component to help accomplishing the task
- In that case, we have an **object within another object**
- Example:



```
class Person {
public:
   Person(const char *name);
   ~Person();
   char *getName() const;
private:
   char *m_name;
};
```

```
class SaleDept {
public:
   SaleDept(const char *manager,
            const char *clerk);
   void listMembers() const;
private:
   Person m_manager;
   Person m_clerk;
};
```

```
void main() {
   SaleDept *saleDept = new SaleDept("Jamie", "Paul");
   saleDept->listMembers();
   delete saleDept;
}
```

error C2512: 'Person' : no appropriate default constructor available

```
SaleDept::SaleDept(const char *managerName, const char *clerkName) {
}
```


Solving The Initialization Problem

- **First try**: **illegal syntax**, calling Person ctor within SaleDept ctor

```
SaleDept::SaleDept(const char *managerName, const char *clerkName) {
   m_manager(managerName);
   m_clerk(clerkName);
}
```

- **Second try**: **still missing default ctor**, require default ctor, invoke assignment operator, depending on some uncertain factors (shallow copy)

```
SaleDept::SaleDept(const char *managerName, const char *clerkName) {
   m_manager = Person(managerName);
   m_clerk = Person(clerkName);
}
```

error C2512: 'Person' : no appropriate default constructor available

The Initialization Problem (cont'd)

- **Third try**: a safe and syntactically legal solution, but **undesirable**

```
class Person {
public:
   …
   Person() {} // empty default ctor
   void setName(const char *name);
};
```

- **Correct solution**: using **initialization list**

```
SaleDept::SaleDept(const char *managerName, const char *clerkName)
   : m_manager(managerName), m_clerk(clerkName) {
}
```


Returning Pointers

```
class Person {
public:
   Person(const char *name);
   ~Person();
   char *getName() const;
private:
   char *m_name;
};
```

- Why? Consider the code:

```
void SaleDept::listMembers() const {
   cout << m_manager.getName()
        << " is the manager of the "
           "sale department and "
        <<  m_clerk.getName()
        << " is the clerk.\n";
}
```

looks OK

- The function **getName()** violates data encapsulation
- What would happen if it were written like this

```
void SaleDept::listMembers() const {
   *m_manager.getName() = '#';
   cout << m_manager.getName() << " is the manager of the sale department and "
        <<  m_clerk.getName() << " is the clerk.\n";
}
```

Interfering the integrity of the private data of Person class

Solving Encapsulation Problem

- **Simple solution** provided by the grammar to prevent incidental breaking of the encapsulation

```
class Person {
public:
   Person(const char *name);
   ~Person();
   const char *getName() const;
private:
   char *m_name;
};
```

unintentional

Won't be able to mutate the content of m_name within this member function

```
const char *Person::getName() const {
   return m_name;
}
```

```
void SaleDept::listMembers() const {
   const char *tempString = m_manager.getName();
   // tempString[0] = '#';   // compiler rejects this statement
   cout << m_manager.getName() << " is the manager of the sale department and "
        <<  m_clerk.getName() << " is the clerk.\n";
}
```

- Other solutions? use a ***string*** object as component

this pointer

- In the first C++ translator, by Stroustrup, C++ functions is translated to pure C functions. How can a function access some variables (those member variables) not defined in that function?

```
class Grades {
public:
   Grades(int score);
   int getScore();
private:
   int m_score;
};
```

```
void main() {
   Grades student1(95), student2(85), student3(45);
   cout << student1.getScore();
   cout << student2.getScore();
   cout << student3.getScore();
}
```

```
int Grades::getScore() {
   return m_score;
}
```

Which variable is this referring to?

- Explicitly access the object

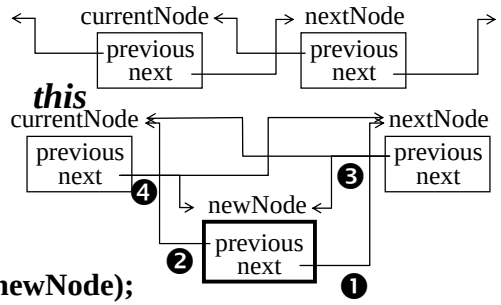
```
int Grades::getScore() {
   return this->m_score;
}
```

- The compiler generates an implicit pointer ***this*** to the object, calls the function, and passes it into the function as an argument.

Primary purpose of *this* pointer

- The ***this*** pointer is most commonly used when objects need to be linked to other objects in a doubly linked list

```
class Node {
public:
   void insert(Node *newNode);
private:
   Node *previous;
   Node *next;
};
```



- We want to insert a new node into the list after currentNode with **currentNode–>insert(newNode);**
- The only way to achieve the goal is using ***this*** pointer

```
void Node::insert(Node *newNode) {
❶ newNode–>next = next;
❷ newNode–>previous = this;
❸ if (next) next–>previous = newNode;
❹ next = newNode;
 }
```

next could be NULL

next->previous

Exploiting Implicit References

- Suppose we want to add a function to class **Grades** that checks if two objects contain the same score
- Here is the call in main()

```
if (grade1.equal(grade2))
   cout << "same scores";
else
   cout << "different scores";
```

- Here is the function

```
bool Grades::equal(Grades &secondScore) {
   return m_score == secondScore.m_score;
}
```

- Do not ignore implicit dereferencing

```
bool Grades::equal(Grades &firstScore, Grades &secondScore) {
   return firstScore.m_score == secondScore.m_score;
}
```

Note how clumsy the call is to this function

```
if (grade1.equal(grade1, grade2))
   ….
```


Type Conversion Constructor

- Suppose we would like to convert raw minutes to Time object

```
class Time {
public:
   Time();
   Time(int hours, int minutes, int seconds);
   Time(int rawMinutes);
private:
   int m_hours;
   int m_minutes;
   int m_seconds;
   void normalize();
};
```

single-argument constructor

```
void Time::normalize() {
   m_minutes += m_seconds / 60;
   m_seconds = m_seconds % 60;
   m_hours += m_minutes / 60;
   m_minutes = m_minutes % 60;
   m_hours = m_hours % 24;
}
```

```
Time::Time(): m_seconds(0), m_minutes(0), m_hours(0) {
}
Time::Time(int hours, int minutes, int seconds)
   : m_hours(hours), m_minutes(minutes), m_seconds(seconds) {
   normalize();
}
Time::Time(int rawMinutes): m_seconds(0), m_minutes(rawMinutes), m_hours(0) {
   normalize();
}
```


Type Conversion Ctor (cont'd)

- Usage:

```
void main() {
   int x = 125;
   Time object;
   object = Time(125);  // temporary object, assignment operator
   object = (Time) x;
   object = 125;
   object = x;
}
```

Explicit invocation of type conversion ctor

Implicit invocation of type conversion ctor, construct a temporary object, execute default assignment operator

- How do we prevent the compiler from using a single-argument constructor in the above implicit conversion?

```
class Time {
   …
   explicit Time(int rawMinutes);
   …
};
```

Class Conversion

```
class Celsius;  // forward declaration
class Fahrenheit {
public:
   Fahrenheit(int temperature);
   Fahrenheit(Celsius &cTemperature);
   int getTemperature() const;
private:
   int m_temperature;
};
class Celsius {
public:
   Celsius(int temperature);
   Celsius(Fahrenheit &fTemperature);
   int getTemperature() const;
private:
   int m_temperature;
};
```

```
Fahrenheit::Fahrenheit(Celsius &cTemperature) {
   int celsiusTemperature = cTemperature.getTemperature();
   m_temperature = (int)(9.0 * celsiusTemperature / 5 + 32.5);
}
```

Usage:

```
Fahrenheit room(75);
Celsius zimmer(18);
Celsius c_room(room);
Fahrenheit f_zimmer(zimmer);
room = zimmer;
```


Static Data Members

- Suppose we want to give each object of the Student class a unique ID
- Using a global variable is one method

// Student.h

```
class Student {
public:
   Student();
   int getID() const;
private:
   int m_id;
};
```

// Student.cpp

```
int gIDNumber = 0;

Student::Student():m_id(gIDNumber++) {
}
```

- Problems:
  - If other programs manipulate this global variable, the count would be incorrect
  - It would be better if a name like gStudentIDNumber is used

Static Data Members (cont'd)

- A better solution with static data member
  - A class declaration is only a type definition instead of a variable, you must define the static variable in the global scope

Student.h

```
class Student {
public:
   Student();
   int getID() const;
private:
   static int lastIDNumber;
   int m_id;
};
```

Student.cpp

```
int Student::lastIDNumber = 0;

Student::Student():m_id(lastIDNumber++) {
}
```

- Also used **for class specific constant** definition

```
class Integer {
   …
   const static INT_MAX = 2147483647;
   …
};
```


Static Member Functions

- Static member functions can only access static data member

```
class Student {
public:
   Student();
   int getID() const;
private:
   static int lastIDNumber;
   int m_id;
   static int getNewID();
   static int incrementNewID();
};
```

- The constructor might take this form

```
Student::Student():m_id(getNewID()) {
   incrementNewID()
}
```

- The keyword **static** is not repeated in the function definitions

```
int Student::getNewID() {
   return lastIDNumber;
}
```

```
int Student::incrementNewID() {
   return lastIDNumber++;
}
```

Static Member Functions (cont'd)

- If the static member function is public, it can be accessed without reference to a particular object, ex.

```
Integer::convertFromInt(10);
Integer::unitTest();
```

- Static member function does not have the implicit *this* pointer because it is not invoked with any object.

- Sometimes use static member functions to implement callback functions that do not allow any implicit argument.