5: Hiding the Implementation

김동원

2003/02/19
Overview

- Object layout
- Handle classes
Object layout

• The class
• Modify Stash to use access control
• Modify Stack to use access control
The class

• **Access control**
  – often referred to as *implementation hiding*
  – Two reasons
    - To establish what the client programmers can and can’t use
    - To separate the interface from the implementation
      - You can change anything that’s private without requiring modifications to their code

• **Including functions within structures**
  – often referred to as *encapsulation*
  – produces a data type with characteristics and behaviors
  – The difference with hiding
    - Access control puts boundaries within that data type
The class

• **Use of class keyword**
  – Identical to the `struct` keyword in absolutely every way except one
    ▪ class defaults to private, whereas struct defaults to public
struct A {
private:
    int i, j, k;
public:
    int f();
    void g();
};
int A::f()
{
    return i + j + k;
}
void A::g()
{
    i = j = k = 0;
}

// Identical results are produced with:
class B {
    int i, j, k;
public:
    int f();
    void g();
};
int B::f()
{
    return i + j + k;
}
void B::g()
{
    i = j = k = 0;
}
int main()
{
    A a;
    B b;
    a.f(); a.g();
    b.f(); b.g();
}
Modifying Stash to use access control

```cpp
#ifndef STASH_H
#define STASH_H

class Stash {
    int size; // Size of each space
    int quantity; // Number of storage spaces
    int next; // Next empty space
    // Dynamically allocated array of bytes:
    unsigned char* storage;
    void inflate(int increase);

public:
    void initialize(int size);
    void cleanup();
    int add(void* element);
    void* fetch(int index);
    int count();
};
#endif
```

It is used only by the `add()` function and is thus part of the underlying implementation, not the interface.
Modifying Stack to use access control

```c++
#ifndef STACK2_H
#define STACK2_H

class Stack {
    struct Link {
        void* data;
        Link* next;
        void initialize(void* dat, Link* nxt);
    }* head;

public:
    void initialize();
    void push(void* dat);
    void* peek();
    void* pop();
    void cleanup();
};
#endif // STACK2_H
```
Handle classes

- Hiding the implementation
- Reducing recompilation
Reducing recompilation

• **In a library header file**
  – It may show strategic information
  – The company doesn’t want available to competitors
  – Security issue
    ▪ You don’t want to expose any clues in a header file that might help people to crack the code
  – Programmers will directly access the private components anyway, using pointers and casting

• **To solve this problem**
  – The actual structure compiled inside an implementation file rather than exposed in a header file
Reducing recompilation

• Cause a recompilation of a file
  – If that file is touched (that is, modified)
  – If another file it’s dependent upon – that is, an included header file – is touched
  – referred to as the *fragile base-class problem*

• The technique to solve
  – This is sometimes called *handle classes* or the *Cheshire cat*
    • everything about the implementation disappears except for a single pointer
Reducing recompilation (example)

```c++
//: C05:Handle.h
#ifndef HANDLE_H
#define HANDLE_H
class Handle {
    struct Cheshire; // Class declaration only
    Cheshire* smile;

public:
    void initialize();
    void cleanup();
    int read();
    void change(int);
};
#endif // HANDLE_H
```
Reducing recompilation (example)

#include "Handle.h"
#include "../require.h"

// Define Handle's implementation:
struct Handle::Cheshire {
  int i;
};

void Handle::initialize()
{
  smile = new Cheshire;
  smile->i = 0;
}

void Handle::cleanup()
{
  delete smile;
}

int Handle::read()
{
  return smile->i;
}

void Handle::change(int x)
{
  smile->i = x;
}
Reducing recompilation (example)

//: C05:UseHandle.cpp
#include "Handle.h"
int main()
{
    Handle u;
    u.initialize();
    u.read();
    u.change(1);
    u.cleanup();
}