第3章　隐 藏 实 现

一个典型的C语言库通常包含一个结构和一组运行于该结构之上的相关函数。前面我们已经看到C++是怎样处理那些在概念上和语法上相关联的函数的，那就是：

把函数的声明放在一个struct内，改变这些函数的调用方法，在调用过程中不再把 struct的地址作为第一个参数传递，在程序中增加一个新的数据类型（这样就不必在 struct关键字前加一个typedef之类的声明了）。

这样做带来很多方便——有助于组织代码，使程序易于编写和阅读。然而，在使得 C++库比以前更容易的同时，存在一些其他问题，特别是在安全与控制方面。本章重点讨论 struct中的边界问题。

3.1　设置限制

在任何关系中，存在相关各方都遵从的边界是很重要的。当我们建立了一个库之后，我们就与该库的用户（也可以叫用户程序员）建立了一种关系，他是另外的程序员，但他需要用我们的库来编写一个应用程序或用我们的库来建立更大的库。

在C语言中，struct同其他数据结构一样，没有任何规则，用户可以在 struct中做他们想做的任何事情，没有办法来强制任何特殊的行为。比如，即使我们已经看到了上一章中提到的initialize()函数和cleanup()函数的重要性，但用户有权决定是否调用它们（我们将在下一章看到更好的方法）。再比如，我们可能不愿意让用户去直接处理 struct中的某些成员，但在C语言中没有任何方法可以阻止用户。一切都是暴露无遗的。

需要控制对结构成员的存取有两个理由：一是让用户避开一些他们不需要使用的工具，这些工具对数据类型内部的处理来说是必须的，但对用户特定问题的接口来说却不是必须的。这实际上是为用户提供了方便，因为他们可以很容易地知道，对他们来说哪些是重要的，哪些是可以忽略的。

二是设计者可以改变 struct的内部实现，而不必担心对用户程序员产生影响。在上一章stack的例子中，我们想以大块的方式来分配存储空间，提高速度，而不是在每次增加成员时调用malloc() 函数来重新分配内存。如果这些库的接口部分与实现部分是清楚地分开的，并作了保护，那么我们只需要让用户重新连接一遍就可以了。

3.2　C++的存取控制

C++语言引进了三个新的关键字，用于在 struct中设置边界：public、private和protected。它们的使用和含义从字面上就能理解。这些存取指定符只在 struct声明中使用，它们可以改变在它们之后的所有声明的边界。使用存取指定符，后面必须跟上一个冒号。

public意味着在其后声明的所有成员对所有的人都可以存取。 public成员就如同一般的struct成员。比如，下面的struct声明是相同的：

```
//: PUBLIC.CPP -- Public is just like C struct

struct A {
```

```
  int i;
  char j;
  float f;
  void foo();
};

void A::foo() {}

struct B {
public:
  int i;
  char j;
  float f;
  void foo();
};

void B::foo() {}
main() {}
```

private关键字则意味着，除了该类型的创建者和类的内部成员函数之外，任何人都不能存取这些成员。private在设计者与用户之间筑起了一道墙。如果有人试图存取一个私有成员，就会产生一个编译错误。在上面的例子中，我们可能想让 struct B中的部分数据成员隐藏起来，只有我们自己能存取它们：

```
//: PRIVATE.CPP -- Setting the boundary

struct B {
private:
  char j;
  float f;
public:
  int i;
  void foo();
};

void B::foo() {
  i = 0;
  j = '0';
  f = 0.0;
};

main() {
  B b;
  b.i = 1;     // OK, public
//!  b.j = '1';  // Illegal, private
//!  b.f = 1.0;  // Illegal, private
}
```

虽然foo()函数可以访问B的所有成员，但一般的全局函数如main()却不能，当然其他struct中的成员函数同样也不能。只有那些在这个struct中明确声明了的函数才能访问这些私有成员。

对存取指定符的顺序没有特别的要求，它们可以不止一次出现，它们影响在它们之后和下一个存取指定符之前声明的所有成员。

保护(protected)

最后一种存取指定符是protected。protected与private基本相似，只有一点不同：继承的结构可以访问protected成员，但不能访问private成员。但我们要到第13章才讨论继承。现在就把这两种指定符当成一样来看待，直到介绍了继承后再区分这两类指定符。

3.3 友元

如果程序员想允许不属于当前结构的一个成员函数存取结构中的数据，那该怎么办呢？他可以在struct内部声明这个函数为友元。注意，一个友元必须在一个 struct内声明，这一点很重要，因为他（和编译器）必须能读取这个结构的声明以理解这个数据类型的大小、行为等方面的规则。有一条规则在任何关系中都很重要，那就是“ 谁可以访问我的私有实现部分 ”。

类控制着哪些代码可以存取它的成员。让程序员没有办法“ 破门而入 ”，他不能声明一个新类然后说“ 嘿，我是鲍勃的朋友 ”，不能指望这样就可以访问鲍勃的私有成员和保护成员。

程序员可以把一个全局函数声明为友元类，也可以把另一个 struct中的成员函数甚至整个struct都声明为友元类，请看下面的例子：

```
//: FRIEND.CPP -- Friend allows special access

struct X; // Declaration (incomplete type spec)

struct Y {
  void f(X*);
};

struct X { // Definition
private:
  int i;
public:
  void initialize();
  friend void g(X*, int); // Global friend
  friend void Y::f(X*);  // Struct member friend
  friend struct Z; // Entire struct is a friend
  friend void h();
};

void X::initialize() { i = 0; }

void g(X* x, int i) { x->i = i; }

void Y::f(X* x) { x->i = 47; }
```

```
struct Z {
private:
  int j;
public:
  void initialize();
  void g(X* x);
};

void Z::initialize() { j = 99; }

void Z::g(X* x) { x->i += j; }

void h() {
  X x;
  x.i = 100; // Direct data manipulation
}

main() {
  X x;
  Z z;
  z.g(&x);
}
```

struct Y有一个成员函数f( )，它将修改X类型的对象。这里有一个难题，因为C++的编译器要求在引用任一变量之前必须声明，所以struct Y必须在它的成员Y :: f(X*)被声明为struct X的一个友元之前声明，但Y :: f(X*)要被声明，struct X又必须先声明。

解决的办法是：注意到Y :: f(X*)引用了一个X对象的地址。这一点很关键，因为编译器知道如何传递一个地址，这一地址大小是一定的，而不管被传递的对象类型大小。如果试图传递整个对象，编译器就必须知道X的全部定义以确定它的大小以及如何传递它，这就使程序员无法声明一个类似于Y :: g(X)的函数。

通过传递X的地址，编译器允许程序员在声明Y :: f(X*)之前做一个不完全的类型指定。这一点是在struct X的声明时完成的，这儿仅仅是告诉编译器，有一个叫X的struct,所以当它被引用时不会产生错误，只要程序员的引用不涉及名字以外的其他信息。

这样，在struct X中，Y :: f(X*)就可以成功地声明为一个友元函数，如果程序员在编译器获得对Y的全部指定信息之前声明它，就会产生一条错误，这种安全措施保证了数据的一致性，同时减少了错误的发生。

再来看看其他两个友元函数，第一个声明将一个全局函数g( )作为一个友元，但g( )在这之前并没有在全局范围内作过声明，这表明friend可以在声明函数的同时又将它作为struct的友元。这种声明对整个struct同样有效:friend struct Z是一个不完全的类型说明，并把整个struct都当作一个友元。

3.3.1 嵌套友元

一个嵌套的struct并不能自动地获得存取私有成员的权限。要获得存取私有成员的权限，必须遵守特定的规则：首先声明一个嵌套的 struct，然后声明它是全局范围使用的一个友元。

struct的声明必须与friend声明分开，否则编译器将不把它看作成员。请看下面的例子：

```
//: NESTFRND.CPP -- Nested friends
#include <stdio.h>
#include <string.h> // memset()
#define SZ 20

struct holder {
private:
  int a[SZ];
public:
  void initialize();
  struct pointer {
  private:
    holder* h;
    int* p;
  public:
    void initialize(holder* H);
    // Move around in the array:
    void next();
    void previous();
    void top();
    void end();
    // Access values:
    int read();
    void set(int i);
  };
  friend holder::pointer;
};

void holder::initialize() {
 memset(a, 0, SZ * sizeof(int));
}

void holder::pointer::initialize(holder* H) {
  h = H;
  p = h->a;
}

void holder::pointer::next() {
  if(p < &(h->a[SZ - 1])) p++;
}

void holder::pointer::previous() {
  if(p > &(h->a[0])) p--;
}
```

```
void holder::pointer::top() {
  p = &(h->a[0]);
}

void holder::pointer::end() {
  p = &(h->a[SZ - 1]);
}

int holder::pointer::read() {
  return *p;
}

void holder::pointer::set(int i) {
  *p = i;
}

main() {
  holder h;
  holder::pointer hp, hp2;
  int i;

  h.initialize();
  hp.initialize(&h);
  hp2.initialize(&h);
  for(i = 0; i < SZ; i++) {
    hp.set(i);
    hp.next();
  }
  hp.top();
  hp2.end();
  for(i = 0; i < SZ; i++) {
    printf("hp = %d, hp2 = %d\n",
           hp.read(), hp2.read());
    hp.next();
    hp2.previous();
  }
}
```

struct holder包含一个整型数组和一个pointer,我们可以通过pointer来存取这些整数。因为pointer与holder紧密相连，所以有必要将它作为struct中的一个成员。一旦pointer被定义，它就可以通过下面的声明来获得存取holder的私有成员的权限：

```
friend  holder : :pointer ;
```

注意，这里struct关键字并不是必须的，因为编译器已经知道pointer是什么了。

因为pointer是同holder分开的，所以程序员可以在main( )块中定义它的多个实例，然后用它们来选择数组的不同部分。由于pointer是C语言中指针的替代，因此程序员可以保证它总是安全地指向holder的内部。

3.3.2 它是纯的吗

这种类的定义提供了有关权限的信息，我们可以知道哪些函数可以改变类的私有部分。如果一个函数被声明为friend，就意味着它不是这个类的成员函数，但却可以修改类的私有成员，而且它必须被列在类的定义中，因此我们可以认为它是一个特权函数。

C++不是完全的面向对象语言，它只是一个混合产品。 friend关键字就是用来解决部分的突发问题。它也说明了这种语言是不纯的。毕竟 C++语言的设计是为了实用，而不是追求理想的抽象。

3.4 对象布局

第2章讲述了为C编译器而写的一个struct，然后一字不动地用C++编译器进行编译。这里我们就来分析struct的布局，也就是，各自的变量放在内存的什么位置？如果C++编译器改变了C struct中的布局，在C语言代码中如果使用了struct中变量的位置信息的话，那么在C++中就会出错。

当我们开始使用一个存取指定符时，我们就已经完全进入了 C++的世界，情况开始有所改变。在一个特定的“ 存取块 ”（被存取指定符限定的一组声明）内，这些变量在内存中肯定是相邻的，这和C语言中一样，然而这些“ 存取块 ”本身可以不按定义的顺序在对象中出现。

虽然编译器通常都是按存取块出现的顺序给它们分配内存，但并不是一定要这样，因为部分机器的结构或操作环境可对私有成员和保护成员提供明确的支持，将其放在特定的内存位置上。C++语言的存取指定并不想限制这种好处。

存取指定符是struct的一部分，它并不影响从这个 struct产生的对象，程序开始运行时，所有的存取指定信息都消失了。存取指定信息通常是在编译期间消失的。在程序运行期间，对象变成了一个存储区域，别无他物，因此，如果有人真的想破坏这些规则并且直接存取内存中的数据，就如在C中所做的那样，那么C++并不能防止他做这种不明智的事，它只是提供给人们一个更容易、更方便的方法。

一般说来，程序员写程序时，依赖特定实现的任何东西都是不合适的。如确有必要，这些指定应封装在一个struct之内，这样当环境改变时，他只需修改一个地方就行了。

3.5 类

存取控制通常是指实现细节的隐藏。将函数包含到一个 struct内（封装）来产生一种带数据和操作的数据类型，但由存取控制在该数据类型之内确定边界。这样做的原因有两个：首先是决定哪些用户可以用，哪些用户不能用。我们可以建立内部的数据结构，而用户只能用接口部分的数据，我们不必担心用户会把内部的数据当作接口数据来存取。

这就直接导出第二个原因，那就是将具体实现与接口分离开来。如果该结构被用在一系列的程序中，而用户只是对公共的接口发送消息，这样程序员就可以改变所有声明为 private的成员而不必去修改用户的代码。

封装和实现细节的隐藏能防止一些情况的发生，而这在 C语言的struct类型中是做不到的。我们现在已经处在面向对象编程的世界中，在这里，结构就是一个对象的类，就像人们可以描述一个鱼类或一个鸟类，任何属于该类的对象都共享这些特征和行为。也就是说，结构的声明开始描述该类型的所有对象及其行为。

在最初的面向对象编程语言 Simula-67中，关键字class被用来描述一个新的数据类型。这显然激发了Stroustrup在C++中选用同样的关键字，以强调这是整个语言的关键所在。新的数据

类型并非只是C中的带有函数的struct，这当然需要用一个新的关键字。

然而class在C++中的使用逐渐变成了一个非必要的关键字。它和struct的每个方面都是一样的，除了class中的成员缺省为私有的，而struct中的成员缺省为public。下面有两个结构，它们将产生相同的结果。

```
//: CLASS.CPP -- Similarity of struct and class

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; }

main() {}
```

在C++中，class是面向对象语言的基本概念，它是一个关键字，本书将不用粗体字来表示。由于要经常用到class，这样做很麻烦。但转换到类是如此重要，我怀疑Stroustrup偏向于将struct重新定义，但考虑到向后兼容性而没有这样做。

许多人喜欢用一种更像struct的风格去创建一个类，因为可以通过以public开头来重载“缺省为私有”的类行为。

```
class X {
public:
  void interface_function();
private:
  void private_function();
  int internal_representation;
};
```

之所以这样做，是因为这样可以让读者更清楚地看到他们的成员是与什么限定符相连的，这样他们可以忽略所有声明为私有成员。事实上，所有其他成员都必须在类中声明的原因仅仅是让编译器知道对象有多大，以便为它们分配合适的存储空间，并保证它们的一致性。

但本书中仍采用首先声明私有成员的方法，如下例：

```
class X {
  void private_function();
  int internal_representation;
public:
  void interface_function();
};
```

有些人甚至不厌其烦地在他们的私有成员名字前加上私有标志：

```
class Y {
public:
  void f();
private:
  int mX;  // "self-mangled" name
};
```

因为mX已经隐藏于Y的范围内，所以在x之前加m并不是必须的。然而在一个有许多全局变量的项目中（有些事虽然我们想极力避免，但有时仍不可避免地出现），它有助于在一个成员函数的定义体内识别出哪些是全局变量，哪些是成员变量。

3.5.1 用存取控制来修改stash

现在我们把第2章的例子用类及存取控制来改写一下。请注意用户的接口部分现在已经很清楚地区分开了，完全不用担心用户会偶然地访问他们不该访问的内容了。

```
//: STASH.H -- Converted to use access control
#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 // STASH_H_
```

inflate()函数声明为私有，因为它只被add()函数调用，所以它属于内在实现部分，不属于接口部分。这就意味着以后我们可以调整这些实现的细节，用不同的系统来管理内存。在此例中除了包含文件的名字之外，只有上面的头文件需要更改，实现文件和测试文件是相同的。

3.5.2 用存取控制来修改stack

对于第二个例子，我们把stack改写成一个类。现在嵌套的数据结构是私有的。这样做的好处是可以确保用户既看不到它，也不能依赖stack的内部表示：

```
//: STACK.H -- Nested structs via linked list
#ifndef STACK_H_
#define STACK_H_

class stack {
  struct link {
    void* data;
    link* next;
    void initialize(void* Data, link* Next);
  } * head;
public:
  void initialize();
  void push(void* Data);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // STACK_H_
```

和上例一样，实现部分无需改动，这里不再赘述。测试部分也一样，唯一改动的地方是类的接口部分的健壮性。存取控制的真正价值体现在开发阶段，防止越界。事实上，只有编译器知道类成员的保护级别，并没有将此类的信息传递给连接器。所有的存取保护检查都是由编译器来完成的，在运行期间不再检查。

注意面向用户的接口部分现在是一个压入堆栈。它是用一个链表结构来实现的，但可以换成其他的形式，而不会影响用户处理问题，更重要的是，无需改动用户的代码。

3.6 句柄类（handle classes）

C++中的存取控制允许将实现与接口部分分开，但实现的隐藏是不完全的。编译器必须知道一个对象的所有部分的声明，以便创建和管理它。我们可以想象一种只需声明一个对象的公共接口部分的编程语言，而将私有的实现部分隐藏起来。但 C++在编译期间要尽可能多地做静态类型检查。这意味着尽早捕获错误，也意味着程序具有更高的效率。然而这对私有的实现部分来说带来两个影响：一是即使程序员不能轻易地访问实现部分，但他可以看到它；二是造成一些不必要的重复编译。

3.6.1 可见的实现部分

有些项目不可让最终用户看到其实现部分。例如可能在一个库的头文件中显示一些策略

信息，但公司不想让这些信息被竞争对手获得。比如从事一个安全性很重要的系统 (如加密算法)，我们不想在文件中暴露任何线索，以防有人破译我们的代码。或许我们把库放在了一个“有敌意”的环境中，在那里程序员会不顾一切地用指针和类型转换存取我们的私有成员。在所有这些情况下，就有必要把一个编译好的实际结构放在实现文件中，而不是让其暴露在头文件中。

3.6.2 减少重复编译

在我们的编程环境中，当一个文件被修改，或它所依赖的文件包含的头文件被修改时，项目负责人需要重复编译这些文件。这意味着无论何时程序员修改了一个类，无论是修改公共的接口部分，还是私有的实现部分，他都得再次编译包含头文件的所有文件。对于一个大的项目而言，在开发初期这可能非常难以处理，因为实现部分可能需要经常改动；如果这个项目非常大，用于编译的时间过多就可能妨碍项目的完成。

解决这个问题的技术有时叫句柄类（handle classes）或叫“Cheshire Cat”[1]。有关实现的任何东西都消失了，只剩一个单一的指针“smile”。该指针指向一个结构，该结构的定义与其所有的成员函数的定义一样出现在实现文件中。这样，只要接口部分不改变，头文件就不需变动。而实现部分可以按需要任意更动，完成后只要对实现文件进行重新编译，然后再连接到项目中。

这里有个说明这一技术的简单例子。头文件中只包含公共的接口和一个简单的没有完全指定的类指针。

```
//: HANDLE.H -- Handle classes
#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_
```

这是所有客户程序员都能看到的。这行

```
struct cheshire;
```

是一个没有完全指定的类型说明或类声明（一个类的定义包含类的主体）。它告诉编译器，cheshire 是一个结构的名字，但没有提供有关该结构的任何东西。这对产生一个指向结构的指针来说已经足够了。但我们在提供一个结构的主体部分之前不能创建一个对象。在这种技术里，包含具体实现的结构主体被隐藏在实现文件中。

---

[1] 这个名字是归属于John Carolan和Lewis Carroll，前者是C++最早的开创者之一。

```
//: HANDLE.CPP -- Handle implementation
#include "..\2\handle.h"
#include <stdlib.h>
#include <assert.h>

// Define handle's implementation:
struct handle::cheshire {
  int i;
};

void handle::initialize() {
  smile = (cheshire*)malloc(sizeof(cheshire));
  assert(smile);
  smile->i = 0;
}

void handle::cleanup() {
  free(smile);
}

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

void handle::change(int x) {
  smile->i = x;
}
```

cheshire 是一个嵌套结构，所以它必须用范围分解符定义

```
struct handle::cheshire {
```

在handle::initialize()中，为cheshire struct分配存储空间 [1]，在handle::cleanup()中这些空间被释放。这些内存被用来代替类的所有私有部分。当编译 HANDLE.CPP时，这个结构的定义被隐藏在目标文件中，没有人能看到它。如果改变了 cheshire的组成，唯一要重新编译的是HANDLE.CPP，因为头文件并没有改动。

句柄（handle）的使用就像任何类的使用一样，包含头文件、创建对象、发送信息。

```
//: USEHANDL.CPP -- Use the handle class
#include "..\2\handle.h"

main() {
  handle u;
  u.initialize();
  u.read();
```

[1] 在第12章我们将看到创建对象更好的方法：用new在堆中分配内存。

```
  u.change(1);
  u.cleanup();
}
```

客户程序员唯一能存取的就是公共的接口部分，因此，只是修改了在实现中的部分，这些文件就不须重新编译。虽然这并不是完美的信息隐藏，但毕竟是一大进步。

3.7 小结

在C++中，存取控制并不是面向对象的特征，但它为类的创建者提供了很有价值的访问控制。类的用户可以清楚地看到，什么可以用，什么应该忽略。更重要的是，它保证了类的用户不会依赖任何类的实现细节。有了这些，我们就能更改类的实现部分，没有人会因此而受到影响，因为他们并不能访问类的这一部分。

一旦我们有了更改实现部分的自由，就可以在以后的时间里改进我们的设计，而且允许犯错误。要知道，无论我们如何小心地计划和设计，都可能犯错误。知道犯些错误也是相对安全的，这意味着我们会变得更有经验，会学得更快，就会更早完成项目。

一个类的公共接口部分是用户能看到的。所以在分析设计阶段，保证接口的正确性更加重要。但这并不是说接口不能作修改。如果我们第一次没有正确地设计接口部分，我们可以再增加函数，这样就不需要删除那些已使用该类的程序代码。

3.8 练习

1) 创建一个类，具有public、private 和protected数据成员和函数成员。创建该类的一个对象，看看当试图存取所有的类成员时会得到一些什么编译信息。

2) 创建一个类和一个全局friend函数来处理类的私有数据。

3) 修改 HANDLE.CPP中的 cheshire，重新编译和连接这一文件，但不重新编译USEHANDL.CPP。