

第五章 类:数据抽象

面向对象程序设计(C++)



5.1 数据抽象

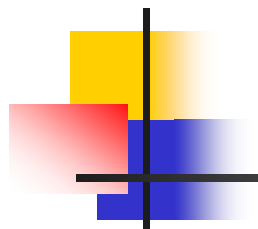
- ➡ C语言的数据描述方法存在的问题
- ➡ 改进：从**Struct**→**Class**的转变
- ➡ 对象与抽象数据类型
- ➡ 对象的细节
- ➡ 头文件的形式
- ➡ 嵌套结构



5.1.1 选择语言的若干因素

- **效率**：该语言实现的程序运行得比较快；
- **安全性**：该语言有助于确信程序实现了我们的意图；同时具有很强的纠错能力。
- **可维护**：该语言能帮助我们创建易理解、易修改和易扩展的代码。
- **成本**：该语言能让我们投入较少的人、在较短的时间内编写出较复杂的、较重要的及无bug的代码。**(核心问题！！)**

对于C语言来说，降低开发成本，提高生产效率的唯一办法是使用“库”（Lib）



5.1.2 什么是C库？

一组struct + 一组作用在这些struct上的函数



5.1.2 一个袖珍的C库: **stash**

- 该库提供一种数据结构stash，对stash的操作像数组，但stash的空间动态申请、动态释放，其长度在运行时确定，并且可以动态扩充。
- 需定义的结构和函数集如下：

```
struct  Stash;  // 结构体Stash

initialize() ;    // 初始化Stash;
cleanup();        // 撤销Stash;
int add();        // 往Stash 添加元素
void* fetch();    // 访问Stash中的元素
int count();      // 返回Stash中的元素个数
void inflate();   // 当Stash满时，动态扩充

Stash
```

库的接口: (CLib.h)



//结构体CStash

```
typedef struct CStashTag {  
    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;  
} CStash;
```

```
void initialize(CStash* s, int size);  
void cleanup(CStash* s);  
int add(CStash* s, const void* element);  
void* fetch(CStash* s, int index);  
int count(CStash* s);  
void inflate(CStash* s, int increase);
```




库的实现:(CLib.cpp)

```
#include "CLib.h"  
#include <iostream>  
#include <cassert>  
using namespace std;
```

```
// Quantity of elements to add when increasing storage  
const int increment = 100;
```

```
void initialize(CStash* s, int sz) { //初始化函数，很重要!  
    s->size = sz;  
    s->quantity = 0;  
    s->storage = 0;  
    s->next = 0;  
}
```



```
int add(CStash* s, const void* element) { //插入元素
    if(s->next >= s->quantity)
        inflate(s, increment); //若空间不足，则动态扩充

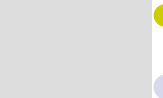
    // Copy element into storage,
    // starting at next empty space:
    int startBytes = s->next * s->size;
    unsigned char* e = (unsigned char*)element;
    for(int i = 0; i < s->size; i++)
        s->storage[startBytes + i] = e[i];
    s->next++;
    return(s->next - 1); // Index number
}

int count(CStash* s) {
    return s->next; // Elements in CStash
}
```




```
void* fetch(CStash* s, int index) { // 返回所需元素的地址
    // Check index boundaries:
    assert(0 <= index);
    if(index >= s->next)
        return 0; // To indicate the end
    // Produce pointer to desired element:
    return &(s->storage[index * s->size]);
}

void cleanup(CStash* s) { // 清除, 回收空间
    if(s->storage != 0) {
        cout << "freeing storage" << endl;
        delete [ ]s->storage;
    }
}
```

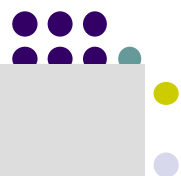


```
void inflate(CStash* s, int increase){  
    //当插入发生溢出时，扩展空间  
    assert(increase > 0);  
    int newQuantity = s->quantity + increase;  
    int newBytes = newQuantity * s->size;  
    int oldBytes = s->quantity * s->size;  
    unsigned char* b = new unsigned char[newBytes];  
    for(int i = 0; i < oldBytes; i++)  
        b[i] = s->storage[i]; // Copy old to new  
    delete [ ](s->storage); // Old storage  
    s->storage = b; // Point to new memory  
    s->quantity = newQuantity;  
}
```

测试/使用该库:



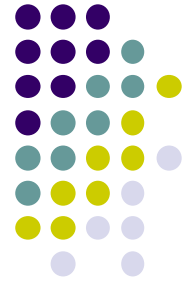
```
// 创建两个Stash:  
// 一个用于存放int; 一个用于存放char[80]  
  
#include "CLib.h"    // 包含库的接口  
#include <fstream>  
#include <iostream>  
#include <string>  
#include <cassert>  
using namespace std;  
  
int main() {  
  
    // Define variables at the beginning  
    // of the block, as in C:  
    CStash intStash, stringStash;
```



```
int i;  
char* cp;  
ifstream in;  
string line;  
const int bufsize = 80;
```

```
// Now remember to initialize the variables:  
initialize(&intStash, sizeof(int)); //显式初始化  
for(i = 0; i < 100; i++)  
    add(&intStash, &i);  
for(i = 0; i < count(&intStash); i++)  
    cout << "fetch(&intStash, " << i << ") = "  
        << *(int*)fetch(&intStash, i)<< endl;
```

```
// Holds 80-character strings:  
initialize(&stringStash, sizeof(char)*bufsize);
```



```
in.open("CLibTest.cpp");
assert(in);
while(getline(in, line))
    add(&stringStash, line.c_str());
i = 0;
while((cp = (char*) fetch (&stringStash,i++))!=0)
    cout << "fetch(&stringStash, " << i << ") = "
        << cp << endl;

cleanup(&intStash);      //显式清除
cleanup(&stringStash);  //显式清除
} ///:~
```



 **思考：**

假设现在你需要创建以下几个stash:
一个用于存放double的Stash,
一个用于存放long的Stash,
还有一个用于存放objA的Stash,
请问你需要在main()中添加哪些代码? 从这里
你认识到C的“库”机制有什么不足?

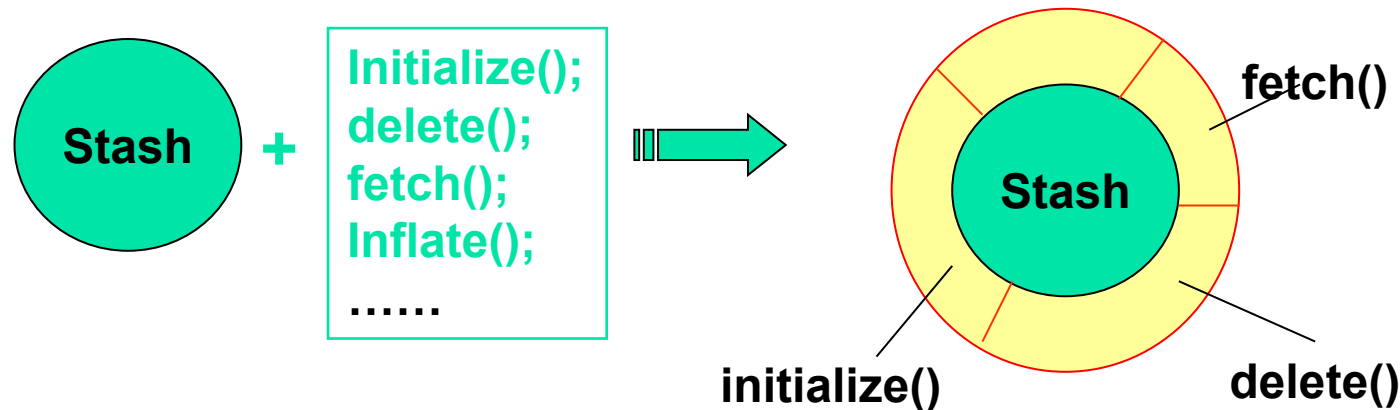


5.1.3 有什么缺陷？

- **用户必须显式初始化和清除，否则：**
 - 内存误用
 - 内存泄漏
- **所有涉及到CStash的文件中必须包含CLib.h，否则：**
 - 编译器不认识Struct Stash;
 - 编译器把未声明的函数调用猜测为一个其他的外部函数调用，往往会“巧合”。
- **不同的库供应厂商由于未协商导致函数名冲突。**

5.1.4 问题出在哪里？

- 不同结构内的同名变量不会冲突，而同名函数会冲突！
 - 因为同名变量被“封装”在结构体内部。而函数没有“封装”。
 - 为什么不把这一优点扩充到在特定结构体上运算的函数上呢？
也就是说，让函数也作为Struct的成员。





5.1.5 迈出第一步：C++的结构

从C的stash → C++的stash：将函数封装到结构中。


```
struct 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;  
  
    // Functions!  
    void initialize(int size);  
    void cleanup();  
    int add(const void* element);  
    void* fetch(int index);  
    int count();  
    void inflate(int increase);  
}; ///~
```

C++结构Stash的实现:

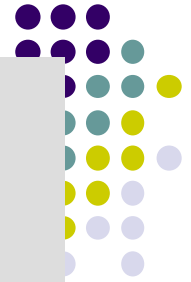


```
// C library converted to C++
// Declare structure and functions:
#include "CppLib.h"
#include <iostream>
#include <cassert>
using namespace std;
/* Quantity of elements to add
   when increasing storage: */
const int increment = 100;

void Stash::initialize(int sz) { /* 注意：函数有明确的从属关
    系；并且不需要显式地传递结构体变量地址（下同） */
    size = sz;
    quantity = 0;
    storage = 0;
    next = 0;
}
```




```
int Stash::add(const void* element) {  
    if(next >= quantity) // Enough space left?  
        inflate(increment);  
    // Copy element into storage,  
    // starting at next empty space:  
    int startBytes = next * size;  
    unsigned char* e = (unsigned char*)element;  
    for(int i = 0; i < size; i++)  
        storage[startBytes + i] = e[i];  
    next++;  
    return(next - 1); // Index number  
}  
  
int Stash::count() {  
    return next; // Number of elements in CStash  
}
```



```
void* Stash::fetch(int index) {  
    // Check index boundaries:  
    assert(0 <= index);  
    if(index >= next)  
        return 0; // To indicate the end  
    // Produce pointer to desired element:  
    return &(storage[index * size]);  
}
```

```
void Stash::cleanup() {  
    if(storage != 0) {  
        cout << "freeing storage" << endl;  
        delete []storage;  
    }  
}
```

```
void Stash::inflate(int increase) {  
    assert(increase > 0);
```



```
int newQuantity = quantity + increase;
int newBytes = newQuantity * size;
int oldBytes = quantity * size;
unsigned char* b = new unsigned char[newBytes];
for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copy old to new
delete []storage; // Old storage
storage = b; // Point to new memory
quantity = newQuantity;
}
```


测试/使用该结构

```
#include "CppLib.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
```

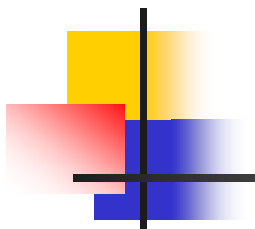


```
using namespace std;
```

```
int main() {  
    Stash intStash;  
    intStash.initialize(sizeof(int));  
    for(int i = 0; i < 100; i++)  
        intStash.add(&i);  
    for(int j = 0; j < intStash.count(); j++)  
        cout << "intStash.fetch(" << j << ") = "  
            << *(int*)intStash.fetch(j) << endl;  
  
    // Holds 80-character strings:  
    Stash stringStash;  
    const int bufsize = 80;  
    stringStash.initialize(sizeof(char) * bufsize);  
    ifstream in("CppLibTest.cpp");  
    assure(in, "CppLibTest.cpp");  
    string line;
```



```
while(getline(in, line))
    stringStash.add(line.c_str());
int k = 0;
char* cp;
while((cp =(char*)stringStash.fetch(k++)) != 0)
    cout << "stringStash.fetch(" << k << ") = "
        << cp << endl;
intStash.cleanup();
stringStash.cleanup();
} ///:~
```



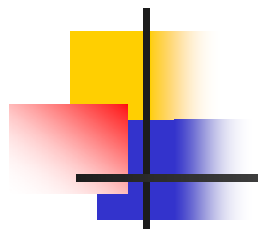
C++结构的特点:

- 函数成了结构的内部成员;
- 实现了封装, 不能任意地增、删结构体内的库函数和数据成员

```
struct Stash{  
    ....; // 数据成员  
    ...; // 成员函数  
}
```

- 用作用域解析运算符” :: ”指示成员的从属关系
- 这样函数有明确的归属

```
void Stash::initialize(int size){  
    ...;  
}
```

C++结构的特点：（续）

- 调用成员函数时，用成员选择符 “.” 指示从属关系
– 由被动转变成主动

```
Stash A;  
A.initialize(80); ...
```

- 对象创建时，只是分配数据成员的存储空间；成员函数的目标代码仍然只有一个拷贝。

```
Stash A,B,C ; //有三个数据区  
A.initialize(80);  
B.initialize(10);  
C.initialize(4); // 共享同一个函数体,
```

在OOP领域中，具有上述特征的结构体变量称为“对象”！

5.1.5 OOP术语：对象

- C中的结构(struct)将数据捆绑在一起，是数据的聚集；如果将专门作用在这些结构上的函数也捆绑到结构中，得到的结构就变成了（基本）对象。
- 对象既能描述属性(attribute)，又能描述行为(behavior)。是一个独立的捆绑的实体(entity)，有自己的记忆(state)和活动。
- 在C++中，对象是“一块存储区”。



Name: J Clark
Employee ID: 567138
Date Hired: July 25, 1991
Status: Tenured





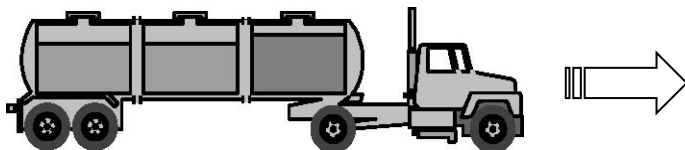
5.1.5 OOP术语：封装

- 将数据连同函数捆绑在一起的能力可以用于创建新的数据类型，通常称这种捆绑为“封装”。

```
struct CStack{  
  
    char * p;           // 属性  
    char * v;           // 属性  
    int size ;          // 属性  
  
    char pop();          // 操作  
    void push(char * c); // 操作  
    char top();          // 操作  
    int size();          // 操作  
}
```

5.1.5 OOP术语：抽象数据类型

- 通过“封装”可以创建一个新的类型；通常是对现实世界的物体的计算机描述，称为“抽象数据类型”。



```
struct truck {  
    // 属性  
    char brand[10];  
    int  speed;  
    ..... ;  
    // 行为  
    start();    // 启动  
    stop();     // 熄火  
    speed_up(); // 加速  
    speed_down(); // 减速  
    turn_left(); // 左转向  
}
```

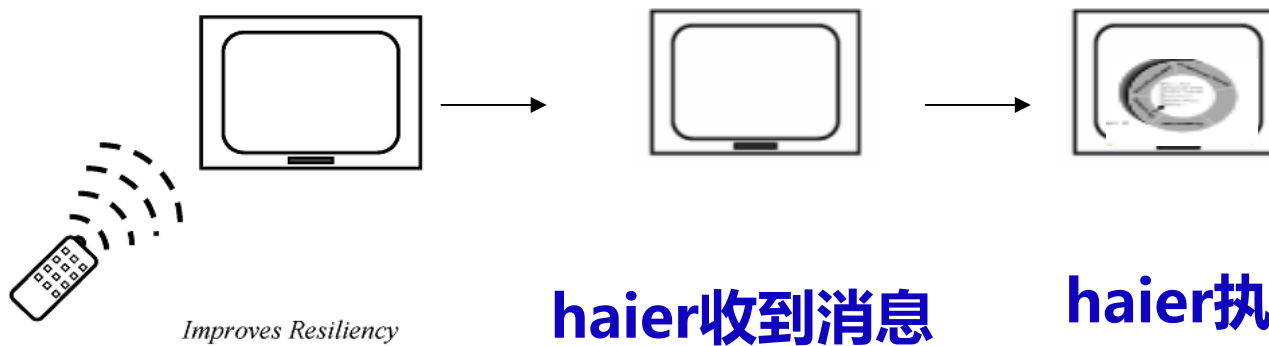


5.1.5 OOP术语：消息

- 对象“调用自己的一个成员函数”，如：
object.member_function(arglist)，是通过对象的使用者完成的，又称之为：使用者“向一个对象发送消息”。当对象收到一个消息后，执行相应的操作。

```
CTV haier;  
// 向对象haier发送消息“turn_on”。  
haier.turn_on();
```

执行过程：🕒



**向对象haier发送
消息" Turn_on";**

haier收到消息

**haier执行
" turn_on"操作**



5.1.6 关于对象的一些细节

- 对象的大小 = 各数据成员大小的和

```
struct A {  
    int i[100];  
    int j;  
    void f();  
};  
  
void main()  
{  
    A aobj;  
}
```

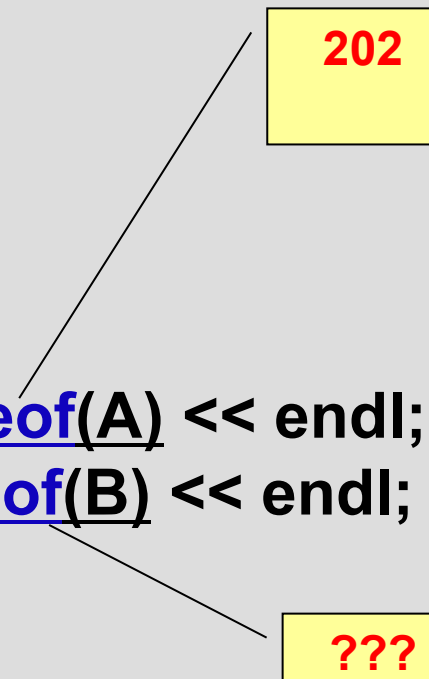
sizeof(aobj)=sizeof(int)*100+sizeof(int)



5.1.6 关于对象的一些细节(续)

- 讨论：当struct不含数据成员时，对象大小=? ?


```
struct B {  
    void f();  
};  
  
void B::f() { }  
  
int main() {  
    B b;   A a;  
    cout << "sizeof A = " << sizeof(A) << endl;  
    cout << "sizeof B = " << sizeof(B) << endl;  
    a.f();  
} ///:~
```



5.1.6 关于对象的一些细节(续)

- 讨论：当struct不含数据成员时，对象大小=? ?

```
struct B {  
    void f();  
};  
  
void B::f() { }  
  
int main() {  
    B b;  A a;  
    cout << "sizeof A = " << sizeof(A) << endl;  
    cout << "sizeof B = " << sizeof(B) << endl;  
    a.f();  
} ///~
```



202

???

1, 用于占位



5.1.7 关于头文件^注

- 头文件中应只限于声明（接口）
（若包含定义，由于多个文件均包含头文件，可能引起重复定义的问题）
- 多次声明的问题：如何避免因多次包含头文件，造成重复声明结构的问题？

```
#ifndef SIMPLE_H
#define SIMPLE_H
struct Simple {
    int i,j,k;
    initialize() { i = j = k = 0; }
};
#endif    //:~ 通常称为 “包含守卫 (include guard) ”
```

注：从软件工程的角度来考察



5.1.7 关于头文件(续)

- **头文件中的名字空间：不要在头文件中使用using指令（后面章节详解）**
- **多个头文件的问题：将头文件、实现和main()函数放在不同的文件中是一种好的习惯。**



5.1.8 嵌套的结构

- **嵌套：**结构的数据成员也可以是一个结构。

```
struct Stack {  
    struct Link {  
        void* data;  
        Link* next;  
        void initialize(void* dat, Link* nxt);  
    }* head;  
    void initialize();  
    void push(void* dat);  
    void* peek();  
    void* pop();  
    void cleanup();  
};
```



```
#include "Stack.h"
#include "../require.h"
using namespace std;
```

```
void Stack::Link::initialize(void* dat, Link* nxt) {
    data = dat;
    next = nxt;
}
```


```
void Stack::initialize() { head = 0; }
```

```
void Stack::push(void* dat) {
    Link* newLink = new Link;
    newLink->initialize(dat, head); // 之前的head成为新
    节点的next
    head = newLink;
}
```


```
void* Stack::peek() {  
    require(head != 0, "Stack empty");  
    return head->data;  
}
```

```
void* Stack::pop() {  
    if(head == 0)    return 0;  
    void* result = head->data;  
    Link* oldHead = head;  
    head = head->next;  
    delete oldHead;  
    return result;  
}
```

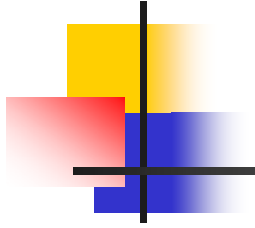
```
void Stack::cleanup() {  
    require(head == 0, "Stack not empty");  
} ///  
~
```



```
// C04:StackTest.cpp  
// Test of nested linked list  
#include "Stack.h"  
#include "../require.h"  
#include <fstream>  
#include <iostream>  
#include <string>  
using namespace std;  
  
int main(int argc, char* argv[ ]) {  
    requireArgs(argc, 1); // File name is argument  
    ifstream in(argv[1]);  
    assure(in, argv[1]);  
    Stack textlines;  
    textlines.initialize();  
    string line;
```

```
// Read file and store lines in the Stack:  
while(getline(in, line))  
    textlines.push(new string(line));  
  
// Pop the lines from the Stack and print them:  
string* s;  
while((s = (string*)textlines.pop()) != 0) {  
    cout << *s << endl;  
    delete s;  
}  
textlines.cleanup();  
} ///:~
```



- 下面两种定义哪种是正确的？

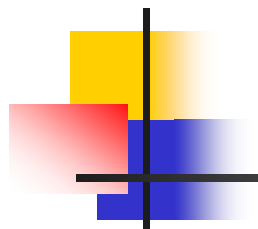
```
struct Link {  
    Link* next;  
};
```

```
struct Link {  
    Link next;  
};
```



小结:

- **数据+函数 称为 “抽象数据类型”**
struct CStash{...};
- **用C++结构创建的变量称为 “对象” （实例） ；**
CStash intStash, floatStash;
- **调用对象的成员函数称为向这个对象发送 “消息” ；**
obj.func();



小结(续):

- **自动初始化：生成对象时设置其数据成员的初值？**
- **自动清除：撤销对象时如何自动回收空间？**
- **访问控制：**
如何控制用户对属性和方法的访问？
 - **什么是用户（结构的使用者）可以改动和访问的？**
 - **什么是程序员（结构的实现者）可以改动和访问的？**