# C 语言重点问题回顾

变量

- 左值和右值

字符串

- 字符串的常见操作
- C 字符串的缺陷
- 自行实现字符串函数的误区

指针和内存

- 指针传递的信息
- 如何正确交换两个变量？--值拷贝问题
- 二维数组传参问题
- malloc 与动态数组、动态二维数组
- 函数指针
- C 语言内存没有属性
- 大小端问题
- 结构体的对齐问题
- 什么是内存泄漏？

线程安全

- 同步、互斥
- 生产者消费者问题　使用条件变量的准则
- C 语言实现线程池

## 左值和右值

左值是指可以位于赋值语句的左边或者右边。

右值，只可以出现在赋值语句的右边。

例如：

```
int i = 5;
const int j = 12;
int &x = i;
```

在上面的代码中，i、x 为左值，j 和 5、12 为右值。注意这里的第二条不是赋值语句！

左值和右值本质区别在哪里？**左值一般用来表明变量的身份，右值则侧重于值本身**。

## 字符串

字符串常见的操作有 strcat、strlen、strcmp 等，C 风格字符串的缺陷就在于需要自己把握内存的大小。

```
char *p;
strcpy(p, "hello");

char str[3];
strcpy(str, "welcome");
```

上面就是两种错误的做法，第一个没有给 p 分配相应的内存空间，第二个则是内存空间不够。

因为以上的原因，很多同学在自己实现字符串的一系列函数中，就出现了以下错误的做法：

```
char *strcat(char *s1, const char *s2){
    s1 = (char *)malloc(strlen(s1) + strlen(s2));
    //

}

char *strcpy(char *s1, const char *s2){

    s1 = (char *)malloc(strlen(s2));
    //
}
```

我们来说明上述代码的错误，首先**对 s1 进行重新分配内存是无效的**，如果我们这样调用 strcat(p, “hello”)，那么 s1 的任何改动均和 p 无关，因为 C 语言传参数采用的是值拷贝，这里实际上是造成了内存泄露。

其次，**就算这里能够改变** p 指向的位置，这样做更加不允许，因为这样会使得其他地方的字符串失效。当然这里不太可能，不做额外讨论。

## 指针和内存

指针本身的含义：

**1.内存的基地址**

**2.数据的类型**

例如 int *p = malloc(100);

char * s = malloc(100);

很显然，p 和 s 本身的值就是内存基地址的数值，但是 p[3]和 s[3]

的值是否相同呢？显然不是，因为 p 和 s 的类型不同，p 是 int 类型指针，所以 p[3]是把后面这段内存当做 int 数组，s[3]则是看做 char 数组。所以 p++一次增加的数值为 4(32bit)，而 s++增加的为 1.

void*是个例外，它只有基地址，没有类型信息，所以无法解引用。

如何正确交换两个变量？--值拷贝问题

```
void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}
```

这段代码究竟错在哪里？

原因在于 C 语言的参数传递方式为 value 拷贝。函数形参拷贝实参的值后，与实参再无关联。

正确的方式是采用指针。这里记住交换变量的原则，**交换 T 类型的变量，那么 swap 函数的形参就要用 T*类型的参数**。也就是说，交换两个 int *，就要使用 int **才可以达到目的。

## 二维数组传参问题

有一个 4*3 的二维数组，我想通过参数传递，在一个函数中打印这个数组，应该怎么办？

新手想到数组可以用 int*传递，二维数组是否可以用 int **传递，于是有下列的代码：

```
void print_array(int **a, int m, int n){
    int i, j;
    for(i = 0; i != m; ++i){
        for(j = 0; j != n; ++j){
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
}
```

我们尝试着编译，使用的是 a[4][3]，得到下列的错误：

error: cannot convert ‘int (*)[3]’ to ‘int**’ for argument ‘1’ to ‘void print_array(int**, int, int)’

这个错误我们可以这样去理解，一维数组可以看做指针 int *，二维数组也是指针，但是不是 int **，而是一维数组的指针。类型为 int(*)[3]，所以引发了错误。

正确的代码是这样的：

```
void print_array(int (*a)[3], int m){
    int i, j;
    for(i = 0; i != m; ++i){
        for(j = 0; j != 3; ++j){
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
}
```

这里大家也能看出这种做法的缺陷，就是我们只能传递第二位为 3 的数组。那么如果才能更通用呢？

我的解决方案是：

1.把数组用一个结构体包装起来。传参数时，传递它的指针。

2.数组用动态内存去分配，这样我们的第一种代码也就是用 int**传递数组是可行的。如何实现动态的二维数组，这就是下面我们要讲述的

内容。

## malloc 与动态数组

二维动态数组的构造方式如下：

```
int **a = (int **)malloc(5 * sizeof(int*));
int i;
for(i = 0; i != 5; ++i){
    a[i] = (int *)malloc(4 * sizeof(int));
}
```

仔细考虑这个数组在内存中的模型，我们先生成一个一维数组，每个元素都是一个指针，然后我们依次为每个指针分配一段内存空间。

考虑以下的问题：

1.这个二维数组能否进行 a[3][3]这样的下标运算？

2.这个数组能否使用 memset 进行初始化？

3.这个数组如何使用 free 进行释放？

## 函数指针

我们过去接触的指针通常是指向变量，但是在 C 语言中，函数也是可以具有指针的。

例如 int *p = &i;

这里是做了两个工作：

1. 我们声明一个 int *类型的指针，这个指针可以指向任何类型的 int

类型变量。

2. 我们用 i 的地址去初始化（**这里是初始化，不是赋值**）p，这样 p 就指向了 i 这个变量。

函数指针也是如此。

对于这样一个函数：

```
void test(int a, int b){
    //
}
```

它的指针是什么类型？

```
void (*) (int , int);
```

这个就是它的指针的类型。

我们使用下列的语句声明一个函数指针变量：

```
void (* pfunc) (int ,int);
```

这里我们声明了一个变量 pfunc，它的类型是 void (*) (int , int)。如何让指针指向一个函数呢？

只需 pfunc = &test 即可。

函数指针的类型看起来比较繁琐，我们尝试用 typedef 进行简化如下：

```
typedef void (* func) (int ,int);
```

这里需要注意的是： **func 是个类型，不是变量**。

## C 语言内存本身没有属性

看下面的代码：

```
char *s = (char *)malloc(1000);
int *p = (int *)s;
```

后面 p 是否能当做普通的 int 数组使用？这段内存是一段“char 数组”，我们把它当 int 使用，例如 p[0]、p[1]，会不会引发问题？

再看另一个例子：

```
struct test{
    int a;
    int b;
};

int main(void) {


    int *p = (int *)malloc(2 * sizeof(int));
    p[0] = 29;
    p[1] = 34;
    struct test * pt = (struct test *)p;
    printf("%d\n", pt->a);
    printf("%d\n", pt->b);
}
```

最后结果打印又是多少？ 原因是为什么？

这里我们解释 C 语言的内存，本质上就是一段 01 区域，**本身是没有任何属性的**，没有所谓的类型 int、char、float 等。我们前面提过指针的本质，提供了两个信息，一个是内存基地址，一个是变量的类型。对于下面的代码：

```
int *p1 = malloc(1000);

char *p2 = malloc(1000);
```

这两段做的内存工作是完全一样的，都是向操作系统申请一块大小为

1000 的内存空间。**并不存在说**，第一块内存是 int 类型，第二块是 char 类型。

那么为什么 p1++和 p2++指针变动的数值不一样？**原因是指针的类型不一样，这与他们指向的内存没有任何关系**。

所以我们得出下列的结论：C 语言中内存都是相同的，如何解释他们，依据的是**采用什么指针操控他们**。

再次举例，对于一段 01 交错的内存区域，用 char*指向他们，我们得到的是 0x55，用 int *指向它们，我们得到的是 0x55 55 55 55.

## 大小端问题

## 结构体的对齐问题

```
struct test{
    int a;
    char c;
};
```

这个结构体在内存中占据几个字节？

## 什么是内存泄漏？

# C++入门

## 第一个 C++程序：

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

## 观察这个程序：

头文件使用的是#include <iostream>

C++标准库中的头文件不需要.h 作为后缀

C 中的头文件能否使用？

仍然可以，以 stdio.h 为例，可以#include <stdio.h>或者#include <cstdio>

用户自定义头文件如何使用？

使用#include “XXX.h”

## 如何运行程序：

如何编译程序？

g++ -o main.o –c main.cpp

如何链接程序？

g++ -o main.exe main.o

## 命名空间：

在这个程序中，使用的是 std::cout 而不是 cout，原因在于 cout 位于标准（std）名称空间中。

那么什么是名称空间呢？

假设调用 cout 时没有使用名称空间限定符，且编译器知道 cout 存在于两个地方，编译器应该调用哪个呢？当然，这回导致冲突，从而导致编译失败。这就是命名空间的用武之地。名称空间是给代码指定的名称，有助于降低命名冲突的风险。通过使用 std::cout，可命令编译器调用名称空间 std 中独一无二的 cout。

如果在代码中频繁添加 std 限定符，会显得很繁琐。为避免添加该限定符，可以使用声明 using namespace std;

如下：

```
#include <iostream>
using namespace std;

int main() {

    cout << "hello world" << endl;

    return 0;
}
```

## 使用 cin 和 cout 执行基本的输入输出操作：

要将简单的文本数据写入到控制台，可使用 std::cout，要从终端读取文本，可以使用 std::cin。

例如：

C++入门 郭春阳

```
#include <iostream>
#include <string>
using namespace std;

int main() {

    int input_number;
    cout << "Please input a number: ";
    cin >> input_number;

    cout << "Enter your name: ";
    string input_name;    //字符串
    cin >> input_name;

    cout << input_name << " " << input_number << endl;
    return 0;
}
```

如何输入未知数目的元素：

```
#include <iostream>
int main()
{
    int sum = 0, value;
    while(std::cin >> value)
    {
    sum += value;
    }
    std::cout << "Sum is: " << sum
              << std::endl;
    return 0;
}
```

练习：

输入一个正整数 n，然后求从 1 到 n 的和，并打印输出。

数据类型：

C++入门 郭春阳

C++新增了 bool 类型来表示真假，C++支持的数据类型有：bool char int float double long 等。

练习：

使用 sizeof 打印每种数据类型的大小

## 左值和右值：

左值：可以放在赋值语句的左边或者右边

右值：只可以放在赋值语句的右边

练习：

举几个左值和右值的例子

## 使用 typedef 简化类型定义

typedef unsigned int UINT;

## const 变量

看下面的代码：

```
#include <iostream>
using namespace std;

int main() {

    const int a = 10;
    a = 33; // ERROR
}
```

## 引用类型：

引用是变量的别名。声明引用时，需要将其初始化为一个变量，

因此引用只是另一种访问相应变量存储数据的方式。

要声明引用，可使用引用运算符（&），如下面的语句所示：

int num = 99;

int &num_ref = num;

当然也可以用于其他类型，例如之前见过的字符串：

string s = "test";

string &s_ref = s;

看下面的程序，观察输出结果：

```
#include <iostream>
using namespace std;

int main() {

    int a = 30;
    cout << "a = " << a << endl;
    cout << "a is at address: " << &a << endl; //打印地址

    int &ref = a;
    cout << "ref is at address: " << &ref << endl;

    int &ref2 = ref;
    cout << "ref2 is at address: " << &ref2 << endl;
    cout << "ref2 gets value, ref2 = " << ref2 << endl;
}
```

输出表明：无论将引用初始化为变量还是其他引用，它都指向相应变量所在的内存单元。因此，引用是真正的别名，即相应变量的另一个名字。

# 数组和字符串

标准库类型

vector

string

## 传统数组的缺点

大小必须在编译时确定，容易溢出

追加元素需要一个 i 记录最后一个位置的下标

不可复制和赋值

在 C 中，我们还可以通过在 heap 上开辟内存的方式来获取动态的数组，这样的好处是可以在运行期决定数组的大小，但是管理麻烦，如果忘了手工 free 内存，就会造成内存泄露。

## vector

C++引入了新的数组 vector，可以克服上面的诸多缺点。

vector 的定义

直接定义一个空数组
也可以指定数组的大小
它可以无限的扩充，不存在溢出问题
支持的操作：
1） 下标访问
2） size（）
3） push_back 从数组的后面追加元素

vector 的使用方法如下：

```
#include <iostream>
#include <vector>
using namespace std;

int main() {

    vector<int> col;  //声明一个空数组

    for(int i = 1; i <= 6; ++i){
        col.push_back(i);//往数组中追加元素
    }

    //遍历打印
    for(int i = 0; i < col.size(); ++i){
        cout << col[i] << " ";
    }
    cout << endl;
}
```

这里我们采用的是声明一个空数组，后面给他追加元素的做法，事实上，这是一种常见的做法。还有一种用法是，一开始就制定好数组的大小，然后逐个进行赋值。

```
#include <iostream>
#include <vector>
using namespace std;

int main() {

    vector<int> col(20);  //数组大小为 20

    for(int i = 0; i != col.size(); ++i){
        col[i] = i;     //根据下标进行赋值
    }

    //遍历打印
    for(int i = 0; i < col.size(); ++i){
```

```
        cout << col[i] << " ";
    }
    cout << endl;
}
```

这里有一些注意点，就是下标不可越界访问，更不可越界赋值，标准中规定越界为非定义行为，引发的后果是未知的！

vector 的另外一种访问方式：迭代器

迭代器是一个“可遍历全部或者部分元素”的对象，但是它的表现行为像是一个指针。这里暂时不介绍迭代器的其他用法。用法如下：

```
#include <iostream>
#include <vector>
using namespace std;

int main() {

    vector<int> col(20);  //数组大小为 20

    for(int i = 0; i != col.size(); ++i){
        col[i] = i;     //根据下标进行赋值
    }

    for(vector<int>::iterator iter = col.begin(); iter != col.end(); ++iter){
        cout << *iter << " ";
    }
    cout << endl;
}
```

后面我们会接触到各种迭代器，但是他们的用法都大同小异。

这里有几处注意点：

对 vector 调用 begin，得到的是一个迭代器，指向 vector 的第一个元素。

对 vector 调用 end，返回一个迭代器，注意，它指向的是容器最后一个元素的下一个位置，也就是说，它指向的是一个不存在的位置。

于是 begin 和 end 就构成了一个半开区间，从第一个元素开始，到最后一个元素的下一个位置结束。半开区间有两个优点：

1.为“遍历元素时，循环的结束时机”提供了一个简单的判断依据。只要尚未到达 end，循环就可以继续进行。

2.不必对空区间采取特殊处理手法。空区间的 begin 等于 end

刚才使用迭代器是正向打印数组，如果想逆向呢，采用下标当然可以，迭代器呢？

```
#include <iostream>
#include <vector>
using namespace std;

int main() {

	vector<int> col(20);  //数组大小为 20

	for (int i = 0; i != col.size(); ++i) {
		col[i] = i;	//根据下标进行赋值
	}

	for (vector<int>::reverse_iterator iter = col.rbegin(); iter != col.rend();
			++iter) {
		cout << *iter << " ";
	}
```

```
    cout << endl;
}
```

这里跟刚才使用迭代器有几处区别：

1.使用的不再是 iterator，而是 reverse_iterator

2.调用的也不再是 begin 和 end，而是 rbegin 和 rend，注意 rbegin 指向最后一个元素，rend 指向第一个元素的前一个位置。

字符串：

C 风格字符串的缺陷：

1.恼人的结束符‘\0’

2.很多时候需要手工保证安全性，例如下面的代码就可能导致程序崩溃。

```
#include <iostream>
#include <string.h>
using namespace std;

int main() {

    char buf[3];
    strcpy(buf, "hello");
    cout << buf << endl;
}
```

这里错误的原因在于 buf 的内存空间不够，于是内存越界，但是 strcpy 不会也无法检查内存是否越界。

C++的解决方案是引入了 string。你可以把 string 当做一个一般的类型去使用，而不会发生任何问题。

C 风格字符串可进行的操作有

strlen 求长度

strcpy 复制字符串

strcmp 比较字符串大小

strcat 连接字符串

这些在 string 中均可以进行，而且更加简单，没有任何的安全隐患。

string 用法如下：

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {

    string s1 = "hello";
    string s2("test");
    string s3;      //空字符串

    cout << s1 << endl; //打印字符串
    cout << s1.size() << endl; //求长度
    s3 = s1;    //字符串间的复制
    s3 += s2;      //字符串的拼接
    cout << s3 << endl;

    //比较大小
```

```
        cout << (s1 < s2) << endl;
        cout << (s1 == s3) << endl;
    }
```

可以看出获取字符串大小使用 string 内部的 size 方法，其他操作采用正常的运算符就可以。

关于相加，这里有几点：

1.string 和 string 可以相加

2.string 和 char*可以相加

3.char*和 string 可以相加

4.char*和 char*不可以相加!!

自行写程序验证

string 同样支持下标操作，所以可以采用下标遍历：

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {

    string s = "who are you?";

    for (size_t ix = 0; ix != s.size(); ++ix) {
        cout << s[ix] << " ";
    }
    cout << endl;
}
```

同样支持迭代器 iterator 和逆置迭代器 reverse_iterator

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {

	string s = "who are you?";

	for (string::iterator iter = s.begin(); iter != s.end(); ++iter) {
		cout << *iter << " ";
	}
	cout << endl;

	for (string::reverse_iterator iter = s.rbegin(); iter != s.rend(); ++iter) {
		cout << *iter << " ";
	}
	cout << endl;
}
```

在 string 中查找字符：

```
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char **argv) {
	string s = "American";

	string::size_type pos = s.find('i');
	if (pos != string::npos) { //查找成功
		cout << pos << endl;
	} else {                        //没有找到
		cout << "not found!" << endl;
	}
}
```

使用的 find 函数，这里要对查找结果进行判断。

string 可以转化为 C 风格字符串：

数组和字符串　郭春阳

```
string s = "Shenzhen";
cout << s.c_str() << endl;
```

注意：string 得出的 char*是 const 属性，也就是说只可以读取它的值，不可以对其进行改动。

另外这个值可能失效，所以如果需要操作字符串，最好是复制一份。

如何整行读入字符串：

使用 getline

代码如下：

```
#include <iostream>
#include <string>
using namespace std;

int main() {
	string s;
	getline(cin, s);
	cout << s << endl;
}
```

# 表达式和异常处理

动态内存分配

new delete

异常处理

try/catch

## 动态内存分配：

在C中我们这样分配内存：

```
int *p = (int*)malloc(100 * sizeof(int));
```

malloc 主要做了两件事情：

1.向系统申请分配内存空间

2.返回该内存空间的首地址

这里注意几点：

1.sizeof（int）申请任何类型变量都采用 sizeof，这是一种跨平台的做法。

2.malloc 返回的是 void*类型的指针，所以这里需要进行强制类型转换。

申请的内存在使用完毕后要进行释放：

```
free(p);
```

free 函数就做了一件事情：

释放 p 指向的内存。

表达式和异常处理 郭春阳

前面说了 malloc 的一些注意点，使用 sizeof 和对结果进行强制类型转换，这也是它的缺点。

C++引入了新的内存分配运算符：new 和 delete

new 的使用方法有三种：

```
int *p1 = new int;
int *p2 = new int(99);
int *p3 = new int[22];
```

下面我们来逐个解释：

1. int *p1 = new int;

申请一个 int 空间，值为默认值

如果这样写 string *ps = new string; 那么就是申请了一个 string，值为空。

2. int *p2 = new int(99);

申请一个 int 空间，值为指定的 99。

string *ps = new string("Beijing");就是动态申请了一个值为 Beijing 的 string。

3. int *p3 = new int[22];

申请一个长度为 22 的 int 数组，值为默认值通常为 0，具体与实现平台相关。

string *ps = new string[10];

ps 指向一个长度为 10 的 string 数组。

表达式和异常处理 郭春阳

new 运算符做了如下的工作：

1.申请内存

2.执行某些类型的构造函数（例如 string）

3.返回这段内存的首地址

那么 new 和 malloc 的区别在哪里？

主要是两点：

1.new 是运算符，malloc 是函数

2.new 会执行某些类型的构造函数，而 malloc 仅仅申请内存。

所以对于 string 类型，只可以使用 new，绝对不可以使用 malloc!!

对应 delete 操作符的使用方法如下：

1.delete ps;

释放 ps 指向的单个变量

所以上面的 p1 和 p2 都应该使用 delete p 的形式

2.delete[] ps;

删除 ps 所指向的数组。所以上面的 ps3 应该用 delete[] 释放

这里我们解释下，**为什么 new 和 delete 的操作要配合使用**，在 C 中我们调用 malloc 申请内存的时候，我们得到一个内存块，在该内存块的**前面**有一处小的区域记录着**这块内存的大小**，所以我们 free 的时

候，系统知道该回收多少内存。同样的道理，我们调用 delete[]，系统也是先去读取前面的区域，获取究竟应该释放多少内存，如果调用 delete，就省略了这一步骤，因为 delete 只需要释放一个单位。这也是 **delete 和 delete[]的本质区别**!

所以，应该使用后者的场合使用了 delete，就造成了内存泄漏。

动态二维数组的申请和释放：

```
int **p = new int*[5];
for (int i = 0; i != 5; ++i) {
    p[i] = new int[4];
}
```

这个数组的特点是：子数组之间连续，数组之间不是连续的。

这个数组仍然支持下标操作，可以自行尝试。

数组的释放过程如下：

```
for (int i = 0; i != 5; ++i) {
    delete[] p[i];
}
delete[] p;
```

注意：

**new 和 delete 的使用必须要配对。**

**前面采用的是 new int[]，后面必须用 delete[]**

**new delete malloc 和 free 不要混用**

## C 语言中的错误处理：

在 C 语言中，我们处理错误最常用的方式是去获取函数调用的**返回值**。在 linux 中，一般用-1 表示函数调用失败，同时 linux 提供一个**全局变量 errno（多线程里面 errno 是线程私有的一个全局变量）**，来记录**错误码**。返回值为 0 或者正数通常表示调用成功。

例如：

```
int fd = socket(...);
if(fd == -1){
    perror("socket:");
    exit(EXIT_FAILURE);
}
```

我们可以**写一个宏**：

```
#define ERR_EXIT(m) \
    do\
    {\
        perror(m);\
        exit(EXIT_FAILURE);\
    }while(0)
```

然后我们就可以这样使用：

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd < 0) {
    ERR_EXIT("socket");
}
```

当然也可以不定义宏，但是定义宏确实减少了代码函数。

## 异常处理：

表达式和异常处理 郭春阳

看一段代码：

```
int fd = socket(...);
if(fd == -1){
    perror("socket:");
    exit(EXIT_FAILURE);
}
```

这段代码用来获取一个 socket，判断获取成功与否，用 fd 是否等于-1 来判断，这种处理错误的方式成为 error code。

C++引入一种**现代编程风格**的错误处理方式：异常处理。

什么是异常处理？首先，异常就是程序运行中的不正常行为。异常处理是采用一种特殊的代码结构对错误进行处理。

例如：

```
int fd;
try {
    fd = socket(...);
    if(fd == -1){
        throw std::runtime_error("socket create error!");
    }

    // .................
} catch (Exception &e) {
    cerr << e.what() << endl; //打印异常信息
}
```

这段代码就是一个异常处理的流程。它的执行流程如下：

1. 创建一个 socket

2. 检查 socket 的返回值为-1，如果不是，一切正常，执行完其他代码，然后直接绕过 catch 块

3. 如果 socket 的返回值为-1，说明出现了错误，这时程序抛出一个异常，这时，程序不再去执行 try 里面其余的代码，而是进入

catch 块，执行里面的错误处理代码。

4. catch 块执行完，程序恢复正常。

从上面的分析可以看出，通常把可能发生异常的代码放入 try 块，把对错误的处理放入到 catch 中。

throw 表达式：

系统通过 throw 抛出异常，如果不对异常进行处理，默认的方式是终止程序。

```
#include <iostream>
#include <stdexcept>
using namespace std;

int main() {

    cout << "before" << endl;
    throw std::runtime_error("test!");

    cout << "after" << endl;

}
```

观察程序的输出可以看到，程序没有输出 after，因为抛出异常时就终止了。

用 try/catch 块捕获异常

```
#include <iostream>
#include <stdexcept>
using namespace std;
```

```
int main() {

    try {
        int num;
        cin >> num;
        if(num == 1)
            throw exception("test1");
        if(num == 2)
            throw exception("test2");

        cout << num << endl;
    } catch (...) {
        cout << "catch a exception" << endl;
    }

}
```

上面的代码中，catch(...)的含义是捕获所有异常。

常见的异常有：

1.exception 最常见的问题

2.out_of_range 越界错误

3.invalid_argument 非法参数

将上面的代码做修改如下：

```
#include <iostream>
#include <stdexcept>
using namespace std;

int main() {

    try {
        int num;
        cin >> num;
        if(num == 1)
            throw out_of_range("test1");
        if(num == 2)
            throw invalid_argument("test2");

        cout << num << endl;
    } catch (...) {
```

```
            cout << "catch a exception" << endl;
        }

    }
```

catch 块仍然正常工作。

用 catch 捕获特性类型的异常

```
#include <iostream>
#include <stdexcept>
using namespace std;

int main() {

    try {
        int num;
        cin >> num;
        if(num == 1)
            throw out_of_range("test1");
        if(num == 2)
            throw invalid_argument("test2");

        cout << num << endl;
    } catch (out_of_range &e) {
        cout << "catch a out_of_range" << endl;
    }

}
```

运行上面的代码，可以看到 catch 块只可以捕获特定类型的异常。如果 try 可能产生不同的类型，可以这样写：

```
#include <iostream>
#include <stdexcept>
using namespace std;

int main() {

    try {
        int num;
        cin >> num;
        if (num == 1)
```

```
            throw out_of_range("test1");
        if (num == 2)
            throw invalid_argument("test2");

        cout << num << endl;
    } catch (out_of_range &e) {
        cout << "catch a out_of_range" << endl;
    } catch (invalid_argument &e) {
        cout << "catch a invalid_argument" << endl;
    }

}
```

在以前的 switch case 语句中，我们通常处理几种特殊情况，其余的采用默认处理，这里也可以这样：

```
#include <iostream>
#include <stdexcept>
using namespace std;

int main() {

    try {
        int num;
        cin >> num;
        if (num == 1)
            throw out_of_range("test1");
        else if (num == 2)
            throw invalid_argument("test2");
        else
            throw exception();
    } catch (out_of_range &e) {
        cout << "catch a out_of_range" << endl;
    } catch (invalid_argument &e) {
        cout << "catch a invalid_argument" << endl;
    } catch (...) {
        cout << "default process" << endl;
    }

}
```

这里有一处值得注意的地方：

```
#include <iostream>
#include <stdexcept>
using namespace std;

int main() {

    try {
        int num;
        cin >> num;
        if (num == 1)
            throw out_of_range("test1");
        else if (num == 2)
            throw invalid_argument("test2");
        else
            throw exception();
    } catch (...) {
        cout << "default process" << endl;
    } catch (out_of_range &e) {
        cout << "catch a out_of_range" << endl;
    } catch (invalid_argument &e) {
        cout << "catch a invalid_argument" << endl;
    }

}
```

上面的程序编译错误，错误信息如下：

error: ‘...’ handler must be the last handler for its try block

原因是 catch(...)可以捕获任意类型的异常，而 catch 块捕获的顺序是从上到下的，一个异常只能被捕获一次。所以后面的 catch 代码均无效。

## 类型转化运算符

C++仍然支持 C 风格的强制类型转换，但是在 C++中，我们更加推荐使用类型安全的转换方式。

C++提供了四种用于类型转换的运算符：static_cast、dynamic_cast、reinterpret_cast、const_cast。

### 1.static_cast

static_cast 用于在相关联的指针类型之间进行转换，还可以显式执行标准数据类型的类型转换。

对 static_cast 类型转换的时机是在编译期间（所以它的名字中含有一个 static），确保指针被转换成相关类型，如果类型不匹配会编译产生错误。这跟 C 显然不一样，C 中的指针可以转化为完全不相关的类型，编译器无法检查出错误。

在 C++中，static_cast 可以用于将派生类的指针向上转型为基类指针，也可以把基类指针向下转换为派生类类型。

使用如下：

```
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char **argv) {

    double pi = 3.1415;
    int val = static_cast<int>(pi);
    cout << val << endl;
```

```
    void *p = new int[8];
    int *ptr = static_cast<int*>(p);
}
```

void*是可以转换为 int*的。但是如果我们在代码末尾加上一句：

```
char *pc = static_cast<char*>(ptr);
```

程序编译便有了问题：error: invalid static_cast from type ‘int*’ to type ‘char*’，这在 C 语言中是检查不出来的。

涉及到基类和派生类的部分我们后面讲解。

2.dynamic_cast

通过名字我们就可以看出来，dynamic_cast 是一种发生在运行期间的强制类型转换。与刚才的 static_cast 不同，前者的检查发生在编译期间，如果转化不合法，那么编译报错。dynamic_cast 如果类型转换不成功，会返回空指针。

注意一点：dynamic_cast 针对的是 class 的类型转换。

3.reinterpret_cast

reinterpret_cast 是 C++中与 C 风格类型转换最接近的类型转换运算符，它让程序员能够将一种对象类型转换为另一种，不管它们是否相关。

实际上，reinterpret_cast 常用来转换 static_cast 不允许的转换风格。例如：

```
#include <iostream>
#include <string>
```

```
using namespace std;

int main(int argc, char **argv) {

    void *p = new int[8];
    int *ptr = reinterpret_cast<int*>(p);

    char *pc = reinterpret_cast<char*>(ptr);
}
```

这段代码是通过编译的。

注意点： reinterpret_cast 与 C 风格类型转换还是有区别的，reinterpret_cast 只改变对指针的操作，但是并不进行**对齐操作**，也就是指针的值绝对不会更改，但是 C 风格类型转换可能会改变指针的值。

4.const_cast

const_cast 可以去掉对象的 const 访问属性。不到万不得已，不要使用它。可以用于转化指针，但是可能导致不可预料的后果。

# 函数

目录：


## 参数传递：

1.pass-by-value（值传递）:

在 C 语言中，参数传递的方式是值传递，例如：

```
int gcd(int v1, int v2) {
    while (v2) {
        int temp = v2;
        v2 = v1 % v2;
        v1 = temp;
    }
    return v1;
}
```

这段代码在函数中修改了 v1 和 v2 的值，假设 main 中这样写：

```
int a = 10;
int b = 25;
cout << gcd(a, b) << endl;
```

函数调用完之后，a 和 b 的值没有发生改变，因为值传递的过程是一个 copy 值的过程，函数的形参在调用的时候是实参的一个副本，它的改动与原来的变量无关。

看一个例子，如何交换两个数，我们知道这样的代码是无效的：

```
void swap(int a, int b){
```

```
    int temp = a;
    a = b;
    b = temp;
}
```

正确的代码是这样的：

```
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

这段代码起作用的原因是拷贝的不是要交换的两个数字，而是两个地址。

注意：**不存在一种地址传递方式叫做指针传递!!!**

练习：　写一个 swap 程序，交换两个 int*  变量。

总结交换两个变量的模板：

假设要交换的变量为 TYPE 类型，那么采用值传递交换需要加一级指针，如下：

```
void swap(TYPE *a, TYPE *b) {
    TYPE temp = *a;
    *a = *b;
    *b = temp;
}
```

看下面的代码：

```
void test(char *p){
    p = new char[100];
}
```

```
int main(int argc, char **argv) {
    char *p = NULL;
    test(p);
    strcpy(p, "hello");
}
```

这段代码存在什么问题？ 如何修正？

1.传递 p 的指针

2.采用返回值

3.采用引用

值传递的缺点：

值传递既然是值拷贝，那么每次函数调用都伴随着变量的复制，如果变量比较大，就会增加程序运行的开销。

2.pass-by-reference（引用传递）:

前面交换两个数字，我们采用了把指针传递进入的办法，这里还可以直接采用引用：

```
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}
```

这段程序之所以能够起到交换的目的，是因为这里采用了引用传递，函数调用时并没有去 copy 两个变量的副本。

采用传递引用的方式交换两个变量的模板为：

```
void swap(TYPE &a, TYPE &b){
    TYPE temp(a);
    a = b;
```

```
        b = temp;
    }
```

引用传递和值传递的区别：

引用传递避免了对象的复制！这很大程度上减少了函数调用的开销。

引用传递的用处：

正因为引用传递有着这样的好处，对于标准库类型变量，我们应该尽可能采用引用作为形参的类型。例如有个函数，输入十个数字，保存在一个数组中，我们可以这样写：

```
void input_num(vector<int> &vec){
    int n = 10;
    int temp;
    while(n--){
        cin >> temp;
        vec.push_back(temp);
    }
}
```

所以，引用形参可以当做输出参数。

采用 const 保护引用形参的值：

如果传递一个字符串，可以这样写：string &s 但是传递引用，使得在函数体内修改 s 的值变为可能（以前值传递是修改的副本，没有副作用），这可能使得代码比较危险，于是我们采用 const 给 string 加上一层保护功能：

```
bool is_short(const std::string &s1, const std::string &s2)
{
    return s1.size() < s2.size();
}
```

这里的 const string &s 有两重含义：

1.避免复制 s，造成开销

2.加上 const 保护 s 的值，防止恶意修改。

实际在编程中，如果一个参数采用的是非 const 引用，这通常意味着**希望**我们修改它的值，如果是 const 引用，那么仅可以读取它的值。

我们总结下**参数传递的原则**：

1.对于原生数据类型，例如 int，如果不需要改变，采用简单的值传递即可，需要改变参数就加上 &符号。

2.对于标准库类型或者用户自定义类型，如果不需要改变，我们使用 const 引用的形式，例如 const A &a，如果需要改变，我们采用 A &a.

## 内联函数：

1.避免了函数调用的开销

2.相对于宏进行了语法检查

注意点：内联函数应该放入到头文件中。

例如我们可以将 swap 写为内联，减少开销，只需在函数开头加上 inline 关键字即可

例如：

```
inline void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}
```

这里可以看做我们写了一个高级的宏函数。

## 函数的返回值：

每条路径都要有返回值：

```
bool str_subrange(const std::string &str1, const std::string &str2)
{
    if(str1.size() == str2.size())
    {
        return str1 == str2;
    }
    std::string::size_type size = (str1.size() < str2.size())? str1.size() : str2.size();
    std::string::size_type i = 0;
    //look at each element up to size of smaller string
    while(i != size)
    {
        if(str1[i] != str2[i])
            return ;
    }
    //error
}
```

上面的代码是错误的，因为有的情况下没有返回值。

返回 value 类型：

例如：

```
string make_plural(size_t ctr, const string &word, const string *&ending){
    return (ctr == 1)? word: word + ending;
}
```

返回的是 string 值，这里返回的是一个副本，因此同样存在对象的复制。

返回 reference 类型：

下列代码的作用是求两个字符较短的一个：

```
const string &shorterString(const std::string &s1,
                            const std::string &s2)
{
    return s1.size() < s2.size() ? s1 : s2;
}
```

这里注意的是：因为 s1 和 s2 都是 const 引用，所以这里返回引用必须加上 const。

再看：

```
char &get_val(std::string &str, std::string::size_type ix)
{
    return str[ix];
}
```

这里返回的就是一个引用。

注意：返回引用的函数可以作为一个**左值**。

例如：

```
char &get_val(std::string &str, std::string::size_type ix)
{
    return str[ix];
}


int main(int argc, char **argv) {
    string s = "hello";
    get_val(s, 2) = 's';
    cout << s << endl;
}
```

**千万不要返回局部对象的引用或者指针！！！**

例如：

```
const std::string &manip(const std::string &s)
{
    std::string ret = s;
    return ret;
}
```

上例中的 ret 是个局部对象，离开这个函数后就被销毁了，此时函数返回一个它的引用，实际上是引用了一块**非法**的内存区域。

## 函数重载：

名称相同，但是参数不同的函数成为函数的重载。在应用程序中，如果需要使用不同的参数调用具有特性名称的函数，函数重载就是很好的解决方案。

```
string sum(const string &a, const string &b){
    return a + b;
}
int sum(int a, int b){
    return a + b;
}
```

这里是两个求和的函数，实际调用时就可以根据参数的类型和个数来辨别应该调用哪个函数。

函数的返回值不能作为函数重载的依据！例如：

```
int sum(int a, int b){
    return a + b;
}

void sum(int a, int b){
    std::cout << (a+b) << std::endl;
}
```

函数的唯一标示包括函数名和形参表，不包括返回值。

后面我们接触到类的成员函数后，函数的唯一标识实际上还可能包含类名和 const 属性。

关于为什么 C 语言没有函数重载，我们到类和对象部分再详细解释。

## 函数指针

下面是一个冒泡排序，可以自定义排序规则：

```
inline void swap(int &a, int &b) {
    int temp(a);
    b = a;
    a = temp;
}

void bubble_sort(vector<int> &vec, bool (*cmp)(int, int)) {
    typedef vector<int>::size_type pos;
    for (pos ix = 0; ix < vec.size() - 1; ++ix) {
        for (pos iy = 0; iy < vec.size() - ix - 1; ++iy) {
            if (cmp(vec[ix], vec[iy])) {
                swap(vec[ix], vec[iy]);
            }
        }
    }
}
```

定义一个函数指针：

```
bool (*pf) (const std::string &, const std::string &);
```

这个函数指针的类型是什么？

```
bool (*) (const std::string &, const std::string &);
```

可以采用这种手段简化函数指针的定义：

```
typedef bool (*cmpFun) (const std::string &, const std::string &);
```

这句话并没有定义一个新的函数指针变量，而是把函数指针的名称简化为 cmpFun

# C++之 IO 流

输入流/输出流

IO 流的状态

文件流

字符串流

C++标准库对 IO 设备进行了抽象，统一使用流的方式对他们进行操作，同时定义了其他类型，例如字符串流，使得 string 对象能够像文件一样操作

前面涉及的 IO 工具有：

istream　　输入流

ostream　　输出流

cin、cout、cerr

>>操作符

<<操作符

getline 整行读取字符串

流的概念：

可以把流看做“水管”，这样输入流指的是从程序外部输入数据，类似于水通过水管流进程序，输出流恰好相反，是从程序内部输出到

外部。这里要注意的是，输入输出的参照物是程序。

典型的输出流是 cin，输出流是 cout 和 cerr。

流的分类：

C++中的 IO 流一共有三种，分别是：

控制台

cin cout cerr

磁盘文件

ifstream

ofstream

fstream

字符串流

istringstream

ostringstream

stringstream

流的特性：

IO 对象不可复制或赋值。

这里有两个含义：一是 IO 对象不能放在 vector 中，而是形参或者返回类型不能为流类型。

所以在使用 IO 流做参数传递或者返回值的时候，统一采用引用的形式，而且一半是非 const 的。

IO 流的条件状态：

考虑下面的程序：

```
int ival;
std::cin >>ival;
```

如果输入 3.14 会发生什么？后面还能继续输入吗？

如果 cin 接收到不符合类型的输入数据，则会读取失败，此时 cin 无法继续读入数据，进入了一个错误状态

如果想继续输入，必须重新设置 cin 流

这就涉及到了 IO 流的三种状态：

bad　系统级故障，不可恢复

fail 可以恢复的错误

eof　cin 碰到了文件结尾

同时系统提供了一系列函数来查询 IO 流的状态

查询状态

```
s.eof()
s.fail()
s.bad()
s.good()      //s 处于有效状态
```

设置状态

```
s.clear()
s.clear(flag)
```

s.setstate(flag)
s.rsstate()

看下面的程序：

```
int a;
if (cin.good()) {
    cout << "cin is good!" << endl;
}
while (cin >> a) {
    cout << a << endl;
}
if (cin.eof()) {
    cout << "eof!" << endl;
}
if (cin.fail()) {
    cout << "fail!" << endl;
}
std::string s;
cin >> s;
cout << s << endl;
```

对于这个程序，我们分别输入如下：

1. 3/4/^D 后面输入 test
2. hello 后面输入 test

我们会看到，对于第一种输入，打印的是：good/eof/fail 以及后面输入的 test，而对于第二个输入，仅仅输出了 good/fail，后面的 test 没有打印出来。

原因是因为：

Cin 接收到非法数据，会导致输入失败，同时 cin 本身不可用！

如果 cin 接收到类型不匹配的数据，则 fail 状态失败，如果接收到 ^D，那么 fail 和 eof 都会失败！

那么流失败的情况下想要继续输入应该怎么办？应该进行修复，示例代码如下：

```
int ival;
using std::cin;
while (cin >> ival, !cin.eof()) {
    if (cin.bad())
        throw std::runtime_error("IO stream corrupted");
    if (cin.fail()) {
        std::cerr << "bad data, try again!" << std::endl;
        cin.clear();    重置状态
        cin.ignore(std::numeric_limits < std::streamsize > ::max(), '\n');        忽略错误输入
        continue;
    }
}
```

当然需要导入相应的头文件

输出缓冲区：

每个 IO 对象都有一个缓冲区，手工刷新缓冲区的办法有：

```
std::cout << "hi" << std::flush;
std::cout << "hi" << std::ends;
std::cout << "hi" << std::endl;
```

在程序中应该多使用 endl，而不是’\n’

文件流

文件流有三种类型

ifstream

ofstream

fstream

打开文件的方式：

两种方式：

```
std::iftream is(“in.txt”)
std::ifstream is;
is.open(filename.c_str());
```

如何从文件中读取文本内容？

```
std::ifstream is;
is.open("in.txt");

std::string word;
std::vector<std::string> vec;
while(is >> word)
{
    vec.push_back(word);
}
```

文件最后要关闭，最好进行 clear 重置状态

从文本进行整行读入：

```
while (std::getline(is, line))
```

如何检测文件是否打开成功？

我们采用下面这个典型的程序：

```
std::ifstream &open_file(std::ifstream &in, const std::string &file) {
    in.close();
    in.clear();
    in.open(file.c_str());
    return in;
}
```

使用方式就是这样：

```
if (!open_file(is, filename)) {
        throw std::runtime_error("file open failed!");
```

```
    }
```

字符串流：

字符串流其实就是把输入输出的对象由文件改为 string，字符串流有下面三种：

istringstream

ostringstream

stringstream

字符串流的使用方式如下：

```
string line, word;
while(getline(cin, line))
{
    istringstream stream(line);
    while(stream >> word)
    {
        //process
    }
}
```

下面编写程序，将文件的每一行存储在 vector<string>中，然后将每个单词存储在另一个 vector<string>中。

代码如下：

```
    std::ifstream & read_file(const std::string &filename, std::ifstream &is,
        std::istringstream &iss, std::vector<std::string> &lines,
        std::vector<std::string> &words) {

    if (!open_file(is, filename)) {
        throw std::runtime_error("file open failed!");
    }
    std::string line;
    std::string word;
    while (std::getline(is, line)) {
        lines.push_back(line);
```

```
            iss.str(line);
            while (iss >> word) {
                words.push_back(word);
            }
            iss.clear();
        }
        is.close();
        is.clear();
        return is;
    }
```

# OOP 之类和对象

--面向对象的第一个特征是：数据抽象

## 目录



## 带有函数的结构体：

前面我们写过这样一个结构体：

```
struct Person{
    int _id;
    string _name;
    int _age;
};
```

在 C 语言中，结构体就是这样定义，只可以含有成员变量，不能含有函数，但在 C++中的结构体可以加上函数。

```
#include <iostream>
#include <string>
using namespace std;

struct Person {
```

```
    int _id;
    string _name;
    int _age;

    void print(std::ostream &os) {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }
};

int main() {

    Person p1;
    p1._age = 99;
    p1._id = 00123;
    p1._name = "Jack";

    p1.print(cout);
}
```

从上面的代码可以看出： 结构体内部的确可以带有函数，而且这些函数可以直接访问里面的数据。

## 两种方式访问对象成员

对于结构体内的成员，包括变量和函数，有两种访问方式：

如果使用的是结构体类型的变量，则采用.的形式。

如果使用的是指针操纵结构体，采用->的形式。例如：

```
#include <iostream>
#include <string>
using namespace std;

struct Person {
    int _id;
    string _name;
    int _age;

    void print(std::ostream &os) {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }
};
```

```
int main() {

    Person p1;
    p1._age = 99;
    p1._id = 00123;
    p1._name = "Jack";
    p1.print(cout);

    Person *p2 = new Person;
    p2->_age = 34;
    p2->_id = 12345;
    p2->_name = "hello";
    p2->print(cout);


}
```

这里提前说明一点：结构体内部的函数包含了一个隐藏的参数，所以这使得这个参数可以直接调用他所在结构体的成员变量。

此时我们的结构体已经具有了成员变量和成员函数，这就是类。

## 类

每个人都有自己的个人信息，其中的有些向周围的人公开，如姓名。这样的信息被成为公有的。然后，有些个人信息可能不想让别人知道，例如收入。这种信息是私有的，通常保密。

C++允许程序员将类的属性和方法设为公有的，这意味着有了对象之后就可以获取他们；也可以将其声明为私有的，这意味着只能在类的内部去访问。C++提供了两个关键字，private 和 public 分别可以把元素设为私有的和公开的。

public：元素是公开的，任何位置都可以访问。

private：成员是私有的，只能在类的内部访问。

我们一般采取的方案是：把类的成员变量设为 private，把类的成员函数设为 public。

上面的结构体中，我们把三个属性设为 private，但是这样一来，三个变量无法访问，于是我们为每个变量提供 get 和 set 方法。

代码如下：

```
#include <iostream>
#include <string>
using namespace std;

class Person {
private:
    int _id;
    string _name;
    int _age;

public:
    int get_id() const {
        return _id;
    }

    void set_id(int id) {
        _id = id;
    }

    string get_name() const {
        return _name;
    }

    void set_name(const string &name) {
        _name = name;
    }

    int get_age() const {
        return _age;
    }

    void set_age(int age) {
```

```
        _age = age;
    }

    void print(std::ostream &os) {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }
};

int main() {

    //编译错误
    Person p1;
    p1._age = 99;
    p1._id = 00123;
    p1._name = "Jack";
    p1.print(cout);

    Person *p2 = new Person;
    p2->_age = 34;
    p2->_id = 12345;
    p2->_name = "hello";
    p2->print(cout);

}
```

很显然代码编译有问题，因为三个属性已经设为 private，不可以直接访问它们。这时只能通过 get 和 set 来访问某个变量。

将 main 里面的内容改为：

```
Person p1;
p1.set_age(23);
p1.set_id(1234);
p1.set_name("hello");
p1.print(cout);
```

此时就没有问题了。

## 成员函数

类的成员函数与其他的函数类似，和任何函数一样，成员函数也包含下面四个部分：

1.函数返回类型

2.函数名

3.形参表

4.函数体

在前面的代码中，我们已经很清楚的看到成员函数如何编写，下面是一些注意点：

1.成员函数含有额外的、隐含的形参

调用成员函数时，实际上是使用对象来调用的。例如上面的 p1.print(cout);，在这个调用中，传递了对象 cout，用 cout 初始化形参 os，但是 print 如何知道打印哪个对象的属性呢？这里实际上把 p1 也作为一个参数传递给了 print。这个隐含的参数将该成员函数和调用该函数的对象捆绑在一起。所以这个函数调用中，print 隐式调用了 p1 这个对象的成员。

2.this 指针的引入

每个成员函数都有一个额外的隐含的形参，这个参数就是 this 指针，它指向调用对象的地址。

在 p1.print(cout)这个代码中，print 中的 this 指针就是 p1 这个对象的地址。

this 指针一般用于解决重名问题和返回自身的值或者引用。例如：

```
struct A{
    int a;

    void test(int a){
        this->a = a;
    }
```

};

test 函数的形参 a 和类成员 a 成名，根据就近原则，直接使用 a，调用的是形参 a，那么如何使用被屏蔽的成员 a 呢，这里就是采用 this 指针。

this 指针的其他用途在本章后面可以看到。

3.const 成员函数

上面我们给每个成员加入了 get 和 set 方法，大家可能已经注意到我们给 get 方法加上了一个 const。这里加 const 的含义是，这个函数不能修改本对象，其实就是函数体内不得对类的成员进行修改。const 主要起到保护的作用。

这里有一点：普通对象可以调用 const 函数，也可以调用非 const 函数，但是 const 对象只能调用 const 函数。

为什么是这样？ 我们不去试图从复杂的语法角度解释，而是联想程序的语义。const 关键字的含义就是**不希望**我们去更改，而不加 const 就是**期望（不一定真的修改，但是程序希望这么做）**我们去修改对象。这样我们在 const 对象里面调用非 const 函数，**我们的意图就显得有些矛盾！**

例如：

```
#include <iostream>
#include <string>
using namespace std;

class Person {
private:
    int _id;
    string _name;
    int _age;
```

```
public:
    int get_id() const {
        return _id;
    }

    void set_id(int id) {
        _id = id;
    }

    string get_name() const {
        return _name;
    }

    void set_name(const string &name) {
        _name = name;
    }

    int get_age() const {
        return _age;
    }

    void set_age(int age) {
        _age = age;
    }

    void print(std::ostream &os) {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }
};



int main() {

    const Person p;
    p.get_age(); //OK
    p.set_age(12); //编译错误!!!!

}
```

请自行查看编译错误的信息。

4. const 成员函数和普通函数可以构成重载

例如：

```
#include <iostream>
#include <string>
using namespace std;

class Person {
private:
    int _id;
    string _name;
    int _age;

public:
    int get_id() const {
        return _id;
    }

    void set_id(int id) {
        _id = id;
    }

    string get_name() const {
        return _name;
    }

    void set_name(const string &name) {
        _name = name;
    }

    int get_age() const {
        return _age;
    }

    void set_age(int age) {
        _age = age;
    }

    void print(std::ostream &os) {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }

    void print(std::ostream &os) const {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }
```

```
};

int main() {

    const Person p1;
    Person p2;
    p2.set_age(12);

    p1.print(cout);
    p2.print(cout);

}
```

这里我们提供了一个 print 的 const 版本，编译通过，说明没有发生函数的重新定义。所以这里的两个 print 构成了重载的关系。

到此为止，构成函数重载的要素有：

**1.函数名**

**2.函数形参表**

**3.类的名称**

4.**成员函数的 const 属性**

事实上，如果你听说过函数签名的概念，那么函数的签名就是由这几个部分构成。

在这里我们解释一个问题： 为什么 C 语言里面没有函数重载？ 在编译器编译程序的时候会维护一张**符号表**，C 语言在记载函数的时候就是简单的记录函数的**名字**，所以名字就是 C 函数的唯一标识。当我们试图定义两个名字相同的函数时，就发生了重定义。

C++是怎么做的呢？ 很显然，对于普通函数，它的符号（**唯一标识**）是根据名字和参数表生成的，对于类的成员函数，还要加上类名和

const 属性，所以我们进行函数重载的时候，这些函数在符号表中的标识是不相同的。 **C++正是通过这种机制实现了函数的重载**。

注意：C++编译器生成函数符号的时候没有考虑返回值，这也是函数重载和返回值无关的原因。

## 构造函数

构造函数是特殊的成员函数，与其他成员函数不同，构造函数与类同名，而且没有返回类型。而与其他成员函数相同的是，构造函数也有形参表（可能为空）和函数体。一个类可以有多个构造函数，每个构造函数必须有与其他构造函数不同数目或类型的形参。

对于 Person 类，可以这样编写构造函数：

```
#include <iostream>
#include <string>
using namespace std;

class Person {
private:
    int _id;
    string _name;
    int _age;

public:

    Person(int id, const string &name, int age){
        _id = id;
        _name = name;
        _age = age;
    }


    int get_id() const {
```

```
        return _id;
    }

    void set_id(int id) {
        _id = id;
    }

    string get_name() const {
        return _name;
    }

    void set_name(const string &name) {
        _name = name;
    }

    int get_age() const {
        return _age;
    }

    void set_age(int age) {
        _age = age;
    }

    void print(std::ostream &os) {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }

    void print(std::ostream &os) const {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }
};

int main() {

    Person p(12, "zhangsan", 12345); //调用我们自己编写的构造函数
    p.print(cout);

}
```

这里注意，构造函数是自动执行，不需要我们去显示调用，而且也不允许手工调用。

1.初始化式

构造函数有一个特殊的地方，就是它可以包含一个构造函数初始化列表：

```
Person(int id, const string &name, int age)
        :_id(id), _name(name), _age(age){
}
```

构造函数初始化列表有个地方难以理解，因为我们这样写也完全可以达到目的：

```
Person(int id, const string &name, int age){
        _id = id;
        _name = name;
        _age = age;
}
```

实际上在有些时候我们必须使用初始化列表：

a)没有默认构造函数的类成员

b)const 成员

c)引用类型的成员

d)有的类成员需要显式调用含参数的构造函数

练习：　自行写程序验证以上的几种类型。

这里有一处陷阱：

考虑下面的类：

```
class X {
    int i;
    int j;
public:
    X(int val) :
            j(val), i(j) {
    }
};
```

我们的设想是这样的，用 val 初始化 j，用 j 的值初始化 i，然而这

里初始化的次序是先 i 然后 j。

记住：类成员初始化的顺序是它们在类中声明的顺序，而不是初始化列表中列出的顺序！

2.默认构造函数

前面我们编写了 Person 类，但是现在这样语句会发生错误：

Person p;

本来这样是正确的，为什么我们自定义了构造函数就错误了？原因是因为如果我们什么都不做，那么编译器自动为我们合成一个默认的无参数的构造函数，类似于：

```
Person() :
                _id(), _name(), _age() {

    }
```

里面每个成员都进行默认初始化。

但是，当我们自行编写了构造函数的时候，编译器就不再为我们提供默认无参数的构造函数，所以我们只能自己提供上面的无参数的构造函数。

当使用 Person *p = new Person[10]; 这段代码生成了 10 个 Person 对象，如果 Person 没有无参数的构造函数，那么这段代码无法通过编译。

练习： 先不为 Person 提供无参数的构造函数，看看编译的错误信息是什么。

3.构造函数的重载

上面我们提供了 Person 的两个构造函数，这就构成了构造函数的

重载。

```
Person() :
        _id(), _name(), _age() {

}

Person(int id, const string &name, int age) :
        _id(id), _name(name), _age(age) {
}
```

最后记住：　构造函数不能为 const。

## 析构函数

与构造函数一样，析构函数也是一种特殊的函数。构造函数在对象被创建时调用，析构函数则是在对象被销毁时被调用。

析构函数看起来有些奇怪，它的名字和类名也相同，只是前面多了一个~，Person 类的析构函数如下：

```
    ~Person(){

        }
```

同样没有返回值，而且析构函数没有任何参数。

## 再谈 this 指针

this 指针最大的作用是返回自身的引用，刚才我们把 Person 的 set 函数返回值设为 void，现在我们改成这样：

```
#include <iostream>
#include <string>
using namespace std;

class Person {
```

```
private:
    int _id;
    string _name;
    int _age;

public:

    Person() :
            _id(-1), _name("none"), _age(-1) {
    }

    Person(int id, const string &name, int age) :
            _id(id), _name(name), _age(age) {
    }

    Person &set_id(int id) {
        _id = id;
        return *this;
    }

    Person &set_name(const string &name) {
        _name = name;
        return *this;
    }

    Person &set_age(int age) {
        _age = age;
        return *this;
    }

    Person &print(std::ostream &os) {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
        return *this;
    }

    const Person &print(std::ostream &os) const {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
        return *this;
    }

};

int main(int argc, char **argv) {
    Person p;
```

```
        p.print(cout);
        p.set_id(12).print(cout).set_age(22).print(cout).set_name("hello").print(
                cout);
}
```

我们做了这么几处改动：将函数的返回值改为返回自身引用，同时为 print 提供了一个重载版本。

函数返回对象自身引用的目的是为了写出:

p.set_id(12).print(cout).set_age(22).print(cout).set_name("hello").print(cout);这样的连续调用式。

如果返回值不加引用，就达不到这样的效果，请自行尝试。

还有一处注意点，const 函数在返回对象成员引用（指针）或者对象自身的引用（指针）时，**必须将返回值设为 const**，否则**造成语意上的矛盾**。

这里很好解释：**const 代表我们不希望进行修改，但是返回一个非 const 引用（它可以作为左值）就为外界修改对象提供了途径，这与 const 的语义是矛盾的。**

这里调用完 print 仍需要进行 set，所以我们提供 print 的两个版本。

## static 静态成员

前面我们提到了成员变量，那么类的每个对象中均存在一个相应的该变量，假设某个类 Point 中含有 x 变量，Point 生成了 100 个对象，那么这 100 个对象均含有一份 x。

如果某个变量属性在某一个类的所有对象中的值均相同，那么把它声明为普通的成员变量，就会造成内存中大量的重复变量。解决方

案是把该变量声明为 static。这样，这个变量不再单独属于某一个对象，而是属于整个类。

```
class   Account{
public:

    Account(){
        ++sum_;
    }
    ~Account(){
        --sum_;
    }

    int show_num();

private:
    static int sum_;   //统计生成了多少对象
};

int Account::sum_ = 0;
int Account::show_num(){
    return sum_;
}
```

上面例子中，sum 是统计 Account 一共生成了多少对象，所以这个数值应该是所有对象集体共享的。

static 变量不与 this 指针绑定

static 变量既然属于整个类，不属于特定的某一个对象，那么它肯定也无法与某一个对象的 this 指针相互关联。这是它与普通变量的重要区别。

成员变量可以声明为 static，函数当然也可以。

上述例子中的 show_num 函数，目的仅仅是打印 sum 这个 static 变量的值，不与任何集体对象发生作用，所以这个函数也可以声明为 static。

static int show_num();

这里注意：**普通成员函数可以访问类的普通成员和 static 成员，但是 static 函数只能访问 static 数据成员**。

## 头文件与前向声明

class Acount; 这个叫做类的前向声明。

如果我们在某个类 B 中使用到了 A 的指针或者引用，那么其实不需要包含完整的头文件（因为这里只需获取指针或者引用的大小），只需要在 B 前面前向声明 A 即可。

如果 B 类包含了 A 对象，或者使用 A 指针（引用）调用了 A 的成员函数，那么就需要 include 头文件。

static 的应用实例：Linux Thread 的封装

其他： Linux MutexLock 和 Condition 的封装

## friend 友元

前面我们把类的数据成员用 private 修饰，这导致我们无法在类的外部直接访问类的数据成员，这显然是一种安全的做法。但很多时候，

类似于现实中的朋友关系，我们往往需要给某些类或者函数提供一些便利，以便于他们可以方便的访问类的成员。

友元类

友元函数

```
class X {
    friend class Y;
    friend void print(const X &x);
private:
    int x_;
    int y_;
};

class Y {
public:
    void print(const X &x) {
        cout << x.x_ << endl;
        cout << x.y_ << endl;
    }
};

void print(const X &x) {
    cout << x.x_ << endl;
    cout << x.y_ << endl;
}
```

在这个例子中，Y 和函数 print 就是类 X 的友元。他们可以直接访问类 X 的成员。

在实际中也有这样一些场景：我们把某个类的所有成员设为 private，只允许它的 friend 成员访问。

## 单例模式

有时候我们需要一个类只能生成唯一的一个对象，这就需要用到一种

设计模式-单例模式。

维基百科上对单例模式的解释如下：

“单例模式，也叫单子模式，是一种常用的软件设计模式。在应用这个模式时，单例对象的类必须保证只有一个实例存在。许多时候整个系统只需要拥有一个的全局对象，这样有利于我们协调系统整体的行为。比如在某个服务器程序中，该服务器的配置信息存放在一个文件中，这些配置数据由一个单例对象统一读取，然后服务进程中的其他对象再通过这个单例对象获取这些配置信息。这种方式简化了在复杂环境下的配置管理。”

实现单例模式的步骤如下：

1.把类的构造函数设为私有，这是为了防止在类的外面随意生成对象。（同时我们最好把类的复制和赋值功能禁用掉），代码如下：

```
class Singleton{
public:

private:
    Singleton(){};

    //forbid copy and assign
    Singleton(const Singleton &);
    Singleton &operator=(const Singleton &);
};
```

2.外面无法生成对象，于是我们尝试在类的内部生成，这里我们添加一个成员函数叫做 getInstance。

```
class Singleton{
public:
    Singleton *getInstance(){
        Singleton *pInstance = new Singleton;
        return pInstance;
    }
private:
    Singleton(){};

    //forbid copy and assign
```

```
    Singleton(const Singleton &);
    Singleton &operator=(const Singleton &);
};
```

但是此时，我们遇到了这样的情况：getInstance 这个函数是用来生成对象的，但是 getInstance 本身又是一个成员函数，需要对象来调用，这造成了相互矛盾。

3.解决上面问题的方法就是把该函数设为 static：

```
#include <iostream>
using namespace std;

class Singleton{
public:
    static Singleton *getInstance(){
        Singleton *pInstance = new Singleton;
        return pInstance;
    }
private:
    Singleton(){};

    //forbid copy and assign
    Singleton(const Singleton &);
    Singleton &operator=(const Singleton &);
};

int main(int argc, char **argv) {
    Singleton *p = Singleton::getInstance();
    cout << p << endl;
}
```

这样我们成功通过这个 static 方法获取了 Singleton 的对象。

4.如何保证对象的唯一性？ 我们添加一个 static 成员变量，每次

生成对象前去检查它是否为 NULL。

```
#include <iostream>
using namespace std;


class Singleton{
public:
    static Singleton *getInstance(){
        //Singleton *pInstance = new Singleton;
        if(pInstance_ == NULL){
            pInstance_ = new Singleton;
        }
        return pInstance_;
    }

private:
    Singleton(){};

    //forbid copy and assign
    Singleton(const Singleton &);
    Singleton &operator=(const Singleton &);


    static Singleton *pInstance_;
};

Singleton *Singleton::pInstance_ = NULL;

int main(int argc, char **argv) {
    Singleton *p1 = Singleton::getInstance();
    Singleton *p2 = Singleton::getInstance();
    cout << p1 << " " << p2 << endl;
}
```

到此为止，我们似乎完成了单例模式的编写，但是上面的程序是否真的可靠？我们编程验证：

测试程序如下：

```
#include <iostream>
#include <vector>
#include <pthread.h>
#include <unistd.h>
using namespace std;

class Singleton {
public:
    static Singleton *getInstance() {
        //Singleton *pInstance = new Singleton;
        if (pInstance_ == NULL) {
            sleep(1);
            pInstance_ = new Singleton;
        }
        return pInstance_;
    }

private:
    Singleton() {
    }
    ;

    //forbid copy and assign
    Singleton(const Singleton &);
    Singleton &operator=(const Singleton &);

    static Singleton *pInstance_;
};

Singleton *Singleton::pInstance_ = NULL;

void *threadFunc(void *arg) {
    Singleton *p = Singleton::getInstance();
    cout << p << endl;
    return NULL;
}

int main(int argc, char **argv) {
    vector<pthread_t> vec(50);
    for (auto &it : vec) {
        pthread_create(&it, NULL, threadFunc, NULL);
    }
    for (auto &it : vec) {
        pthread_join(it, NULL);
```

```
        }
    }
```

我们发现，在多线程的环境中，单例模式编写是失败的，如何解决？ 利用互斥锁 MutexLock。

```
static Singleton *getInstance() {
        //Singleton *pInstance = new Singleton;
        lock_.lock();
        if (pInstance_ == NULL) {
            sleep(1);
            pInstance_ = new Singleton;
        }
        lock_.unlock();
        return pInstance_;
    }
```

上面的程序还有一些不完善的地方，例如每次调用 getInstance 都要上锁，开销太大。我们做一些轻微的改动如下：

```
static Singleton *getInstance() {
    //Singleton *pInstance = new Singleton;
    if (pInstance_ == NULL) {
        lock_.lock();
        if (pInstance_ == NULL) {
            pInstance_ = new Singleton;
        }
        lock_.unlock();
    }
    return pInstance_;
}
```

上面的程序就是经典的“**双重锁**”模式。这种模式在某些情况下也有缺陷，但这超出了我们课程的范畴，在此不做讨论。

# STL 标准模板库

标准模板库（英文：Standard Template Library，缩写：STL），是一个 C++软件库，也是 C++标准程序库的一部分。其中包含 5 个组件，分别为算法、容器、迭代器、函数、适配器。

模板是 C++程序设计语言中的一个重要特征，而标准模板库正是基于此特征。标准模板库使得 C++编程语言在有了同 Java 一样强大的类库的同时，保有了更大的可扩展性。

目录：

Vector 和 list 的区别

String 类型

Stack 和 queue

## 一 顺序容器的初始化

顺序容器主要是 vector 和 list

他们的初始化方式有五种

1.直接初始化一个空的容器

2.用一个容器去初始化另一个容器

3.指定容器的初始大小

4.指定容器的初始大小和初始值

5.用一对迭代器范围去初始化容器

例子如下：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

using namespace std;

int main(int argc, char **argv) {
    vector<string> vec;  //构造一个空数组
    vec.push_back("hello");
    vec.push_back("world");

    vector<string> vec2(vec); //用一个容器去初始化另一个容器

    vector<string> vec3(vec.begin(), vec.end()); //用一对迭代器去遍历另一个容器

    vector<string> vec4(10, "test"); //指定容器的大小和初始值
```

```
    list<string> lst(vec.begin(), vec.end());

    return 0;
}
```

第二种和第五种初始化方式的区别在于：

第二种不仅要求容器类型相同，还要求容器元素类型完全一致，而第五种不要求容器相同，对于容器元素，要求能相互兼容即可

指针可以当做迭代器，所以可以这样做：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

using namespace std;

int main(int argc, char **argv) {
    const size_t MAX_SIZE = 3;
    string arr[MAX_SIZE] = { "hello", "world", "foobar" };

    vector<string> vec(arr, arr + MAX_SIZE);

    return 0;
}
```

## 二 容器元素的类型约束

1.容器元素必须支持赋值运算

2.元素类型的对象必须可以复制

例如：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

using namespace std;

class Student{
private:
    //编译错误的根源所在
    Student(const Student &);
    Student &operator=(const Student &);
};

int main(int argc, char **argv) {
    vector<Student> vec(10);

    return 0;
}
```

这个例子编译时错误的，因为 Student 是不可复制的，不满足 STL 中容器元素的基本要求。

## 三 迭代器

迭代器支持以下操作

```
*iter
Iter->
++iter
--iter
Iter1 == iter2
Iter1 !=iter2
```

Vector 还提供了以下操作

```
Iter+n
Iter1 += iter2
> >= < <=
```

## 迭代器范围

C++ 语言使用一对迭代器标记迭代器范围(iterator range),这两个迭代器分别指向同一个容器中的两个元素或超出末端的下一位置,通常将它们命名为 first 和 last,或 beg 和 end,用于标记容器中的一段元素范围。

尽管 last 和 end 这两个名字很常见,但是它们却容易引起误解。其实第二个迭代器从来都不是指向元素范围的最后一个元素,而是指向最后一个元素的下一位置。该范围内的元素包括迭代器 first 指向的元素,以及从 first 开始一直到迭代器 last 指向的位置之前的所有元素。如果两个迭代器相等,则迭代器范围为空。

此类元素范围称为左闭合区间(left-inclusive interval),其标准表示

方式为:

[ first, last )

表示范围从 first 开始,到 last 结束,但不包括 last。迭代器 last 可以等于 first,或者指向 first 标记的元素后面的某个元素,但绝对不能指 first 标记的元素前面的元素。

## 使用左闭合区间的编程意义

因为左闭合区间有两个方便使用的性质,所以标准库使用此烦区间。

假设 first 和 last 标记了一个有效的迭代器范围,于是:

1. 当 first 与 last 相等时,迭代器范围为空;

2. 当 first 与不相等时,迭代器范围内至少有一个元素,而且 first 指向该区间中的第一元素。此外,通过若干次自增运算可以使 first 的值不断增大,直到 first == last 为止。

这两个性质意味着程序员可以安全地编写如下的循环,通过测试迭代器处理

一段元素:

```
while (first != last) {
   ++first;
}
```

**特殊的迭代器成员 begin 和 end**

有四个特殊的迭代器:

c.begin()  //指向容器 C 的第一个元素

C.end()  //指向最后一个元素的下一个位置

C.rbegin()  //返回一个逆序迭代器，指向容器 c 的最后一个元素

C.rend() //返回一个逆序迭代器，指向容器 c 的第一个元素的前面的位置

分别去顺序迭代和逆序迭代容器，例如:

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

using namespace std;

int main(int argc, char **argv) {
```

```
        vector<string> vec;
        vec.push_back("beijing");
        vec.push_back("shanghai");
        vec.push_back("guangzhou");
        vec.push_back("shenzhen");

        for (vector<string>::iterator iter = vec.begin(); iter !=
    vec.end();
                ++iter) {
            cout << *iter << endl;
        }

        for (vector<string>::reverse_iterator iter = vec.rbegin();
                iter != vec.rend(); ++iter) {
            cout << *iter << endl;
        }

        return 0;
    }
```

## 顺序容器的操作

每个顺序容器都提供了以下操作

添加元素

在容器中删除元素

设置容器大小

容器内置了一些类型，例如 size_type 和 iterator

Size_type：无符号整数，当我们采用下标的方式遍历容器，尽可能采用这种类型

Iterator：此容器类型的迭代器类型，每个容器都内置了自己的迭代

器类型

往容器中添加元素

1.在容器的指定位置添加元素

在容器中使用 push_back 或者 push_front 可以在容器的尾部和首部添加元素

还可以使用 insert(p, t) 在指定位置元素之前添加元素 p 是迭代器，t 时元素的值

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

using namespace std;

int main(int argc, char **argv) {

    vector<string> vec;
    vec.push_back("beijing");
    vec.push_back("shanghai");
    vec.push_back("guangzhou");
    vec.push_back("shenzhen");

    list<string> lst;

    lst.push_front("test"); //在容器的开头添加元素
    lst.push_back("foo");   //在容器后面追加元素

    lst.insert(lst.begin(), "shenzhen");  //在指定迭代器位置添加元素

    return 0;
}
```

2.插入一段元素

插入一段元素采用的是 insert 的另外两个版本

Insert(p, n, t) //在迭代器 p 指向的位置插入 n 个元素，初始值为 t

Insert(p, b, e) //在迭代器 p 指向的位置插入 b 和 e 之间的元素

例如：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

using namespace std;

int main(int argc, char **argv) {

    vector<string> vec;
    vec.push_back("beijing");
    vec.push_back("shanghai");
    vec.push_back("guangzhou");
    vec.push_back("shenzhen");

    list<string> lst;

    lst.insert(lst.begin(), 5, "hangzhou");   //在开头处插入5个“杭州”
    lst.insert(lst.end(), vec.begin(), vec.end()); //在后面添加 vec 的所有元素

    return 0;
}
```

3.迭代器的失效问题

任何 insert 或者 push 操作都可能导致迭代器失效。当编写循环将元素插入到 vector 或 list 容器中时，程序必须确保迭代器在每次循环后都得到更新

4.元素的访问

遍历容器的方式有两种，一种是采用下标，一种是迭代器。

但是 list 不支持下标访问，所以迭代器是更通用的操作

例如：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
using namespace std;
int main(int argc, char **argv) {

    vector<string> vec;
    vec.push_back("beijing");
    vec.push_back("shanghai");
    vec.push_back("guangzhou");
    vec.push_back("shenzhen");

    for(vector<string>::size_type ix = 0; ix != vec.size(); ++ix){
        //
    }

    for(vector<string>::iterator iter = vec.begin(); iter != vec.end();
++iter){
        //
    }

    return 0;
}
```

删除元素

1.删第一个或最后一个元素

类似与插入元素，pop_front 或者 pop_back 可以删除第一个或者最后一个元素

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
using namespace std;
int main(int argc, char **argv) {

    vector<string> vec;
    vec.push_back("beijing");
    vec.push_back("shanghai");
    vec.push_back("guangzhou");
    vec.push_back("shenzhen");

    list<string> lst(vec.begin(), vec.end());

    lst.pop_back();  //删除最后一个元素
    lst.pop_front(); //删除第一个元素

    return 0;
}
```

2.删除容器的一个元素

与 insert 对应，删除采用的是 erase 操作，该操作有两个版本：删除由一个迭代器指向的元素，或者删除由一对迭代器标记的一段元素。

例如：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;
int main(int argc, char **argv) {

    vector<string> vec;
    vec.push_back("beijing");
    vec.push_back("shanghai");
    vec.push_back("guangzhou");
    vec.push_back("shenzhen");
    vec.push_back("jinan");

    list<string> lst(vec.begin(), vec.end());


    //查找“shanghai”元素
    list<string>::iterator  iter1  =  find(lst.begin(),  lst.end(),
"shanghai");
    lst.erase(iter1);//删除这一个元素

    //查找“guanzghou”元素的迭代器和“shenzhen”的迭代器
    list<string>::iterator  first  =  find(lst.begin(),  lst.end(),
"guanzghou"),
            last = find(lst.begin(), lst.end(), "shenzhen");

    lst.erase(first, last); //删除这一段元素

}
```

这里采用了标准库的 find 算法，下文将会提到

删除元素时可能的陷阱：

3.删除容器的所有元素

删除所有元素有两种办法，一种是 clear，一种是采用 erase 删除所有的范围

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;
int main(int argc, char **argv) {

    vector<string> vec;
    vec.push_back("beijing");
    vec.push_back("shanghai");
    vec.push_back("guangzhou");
    vec.push_back("shenzhen");
    vec.push_back("jinan");

    list<string> lst(vec.begin(), vec.end());

    lst.clear();
    lst.erase(lst.begin(), lst.end());
}
```

查找元素的操作

因为顺序容器内部没有内置 find 操作，所以我们采用标准库提供的一个泛型算法：

例子如下：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;
int main(int argc, char **argv) {

    vector<string> vec;
    vec.push_back("beijing");
    vec.push_back("shanghai");
    vec.push_back("guangzhou");
    vec.push_back("shenzhen");
    vec.push_back("jinan");

    std::string query;  //要查询的单词
    vector<string>::iterator iter = std::find(vec.begin(), vec.end(),
           "beijing");
    if (iter == vec.end()) {//表示没有找到
        cout << "not found!" << endl;
    } else {
        cout << *iter << endl;
    }

}
```

需要注意的是，必须考虑到查找不到的情况

容器大小的操作

size()  //返回容器的大小

empty()  //返回容器是否为空

resize(n)  //重置容器的大小

例如：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;
int main(int argc, char **argv) {

    list<string> lst(10);
    cout << lst.size() << endl;

    cout << lst.empty() << endl;

    lst.resize(100); //重置容器元素个数为100
    cout << lst.size() << endl;

}
```

Vector 的内存增加策略

Vector 内部的元素时连续存储的，类似于数据结构中的顺序表

Vector 内置了两个函数：

Capacity 操作获取容器需要分配更多的存储空间之前能够存储的元素个数总数

Reserve 操作告诉 vector 容器应该预留多少个元素的存储空间

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
```

```
using namespace std;
int main(int argc, char **argv) {

    vector<int> vec; //空数组
    cout <<"size:" <<vec.size() <<" capacity:" <<vec.capacity()
<< endl;

    //放入 24 个元素
    for(vector<int>::size_type ix = 0; ix != 24; ++ix){
        vec.push_back(24);
    }
    cout <<"size:" <<vec.size() <<" capacity:" <<vec.capacity()
<< endl;

    //将容器的预留空间设为 50
    vec.reserve(50);
    cout <<"size:" <<vec.size() <<" capacity:" <<vec.capacity()
<< endl;

    //将容器的现有空间填充完毕
    while(vec.size() != vec.capacity()){
        vec.push_back(0);
    }
    cout <<"size:" <<vec.size() <<" capacity:" <<vec.capacity()
<< endl;

    //此时容器满，再放入一个元素
    vec.push_back(42);
    cout <<"size:" <<vec.size() <<" capacity:" <<vec.capacity()
<< endl;

}
```

Vector 和 list 的区别

Vector 内部 采用顺序表实现，而 list 采用链表实现，所以二者的差别很大程度是顺序表和链表的差别

如果不知道选择哪种容器，通常 vector 是最佳选择

String 类型

构造 string 类型的方法

可以使用字符串字面值直接初始化字符串

插入元素

提供了 insert 方法，有大量的重载版本

删除元素

Erase 方法

```
int main(int argc, char **argv) {
    string s = "helloworld";

    s.insert(2, 5, 's');   // pos n  value
    cout << s << endl;

    s = "helloworld";

    s.insert(4, "test");   // pos  C-style string
    cout << s << endl;

    s = "helloworld";
    s.insert(2, "test", 3);  // pos char* len (0 -len-1)
    cout << s << endl;

    s = "helloworld";
    string _tmp = "foobar";
    s.insert(5, _tmp, 3, 2); //pos string pos2 len
    cout << s << endl;  // hellobaworld

    s = "helloworld";
    string _tmp2 = "foobar";
    s.insert(0, _tmp2);   // pos  string
    cout << s << endl;   // foobarhelloworld
```

```
    string s2 = "helloworld";

    s2.erase(4, 5);   //pos n

    cout << s2 << endl;  //helld

}
```

截取字串 substr

Append

在 string 后面串接新的字符串

```
#include "test_include.h"
using namespace std;
int main() {
    string s = "helloworld";

    cout << s.substr(5, 3) << endl;  // pos n

    s.append("hello");

    cout << s << endl;

    s.append("hello", 3); //hel

    cout << s << endl;

    s.append(string("world"));
    cout << s << endl;

    s.append(string("foobar"), 3, 3);  // bar

    cout << s << endl;

    s.append(8, 's');  // 8 s
    cout << s << endl;

    string tmp = "zhangsan";
```

```
    string::iterator it1 = find(tmp.begin(), tmp.end(), 'h'),
           it2 = find(tmp.begin(), tmp.end(), 'g');
    //查找 h 和 g 所在位置的迭代器

    s.append(it1, it2);   //han

    cout << s << endl;


}
```

Replace

将字符插入到某些位置，并替换已经存在的字符

```
#include "test_include.h"
using namespace std;
int main() {
    string s = "helloworld";


    s.replace(4, 3, "test");  // owo -> test

    cout << s << endl;  //helltestrld


    s.replace(4, 3, string("foobar"), 3, 3);   // tes -> bar

    cout << s << endl;  // hellbartrld

}
```

String 内部的查找操作

```
#include "test_include.h"
using namespace std;
int main() {
    string s = "helloworld";

    string::size_type pos1 = s.find(string("test"), 2); //

    if (pos1 != string::npos) {   //如果查找不到
        cout << pos1 << endl;
    } else {
        cout << "not found" << endl;
    }
}
```

String 的比较操作

采用的是字典排序，规则与 C 风格字符串相同

Stack 和 queue

Stack 和 queue 是标准库对栈和队列两种数据结构的封装，他们的使用方法和以前封装的 stack 和 queue 类似

# 关联容器

概述：

目录：

# Pair 类型

Pair 是一种简单的关联类型

创建/初始化/操作如下：

```
#include "test_include.h"

using namespace std;


int main(int argc, char **argv) {

	std::pair<int, string> p1;	// default
	std::pair<int, int> p2(4, 5);
	std::pair<string, string> p3("hello", "world");
	p1.first = 1;
	p1.second = "test";

	cout << p2.first << " " << p2.second << endl;
	cout << p3.first << " " << p3.first << endl;
	make_pair(1, 2);

	make_pair(string("hello"), 1);	// string , int
	vector<string> v1;
	vector<list<string> > v2;
	make_pair(v1, v2);
}
```

# Map 类型

Map 可以看做是一种 pair 的容器，内部时采用二叉树实现的（具体些是红黑树）

# Map 的定义方式：

```
#include "test_include.h"
#include <stack>
#include <queue>
using namespace std;

class Student
{
private:
	int _num;
	string _name;
};

int main(int argc, char **argv) {

	// map    key    value

	std::map<int ,int> m1;

	std::map<string, int> m2;
	map<string, string> m3;


	map<string, vector<string> > m4;

	map<list<vector<list<string> > >, stack<queue<int> > > m5;

	map<int, Student> m6;
	map<Student, int> m7;
}
```

# Map 的遍历

```
#include "test_include.h"
#include <stack>
#include <queue>
```

```
#include <typeinfo>
using namespace std;

//void print(const map<string, int>::value_type &p)
void print(const pair<string, int> &p)
{
	cout << p.first << " " << p.second << endl;
}

int main(int argc, char **argv) {
	map<string, int> people;

	people["shenzhen"] = 1000;
	people["beijing"] = 3000;
	people["shanghai"] = 2000;


	cout << people["beijing"] << endl;

	people["beijing"] = 8000;    // OK

	//遍历这个 map
	map<string, int>::iterator iter = people.begin();
	while (iter != people.end()) {
		cout << iter->first << " " << iter->second << endl;
		++iter;
	}

	cout << "----------" << endl;

	for_each(people.begin(), people.end(), print); //这里用的是标准库算法
}
```

# Map 使用下标访问的一些问题

```
	#include "test_include.h"
	#include <stack>
	#include <queue>
	#include <typeinfo>
	using namespace std;

	int main(int argc, char **argv) {
		map<string, int> word_count;
```

```
    cout << word_count.size() << endl;    // 0

    word_count["hello"];

    cout << word_count.size() << endl; // 1

    word_count["hello"];

    cout << word_count.size() << endl;    //1

    word_count["world"];

    cout << word_count.size() << endl;    //2
}
```

每当用下标去访问 map 元素的时候，如果该元素不存在，那么首先在 map 中新生成一个键值对。所以用下标访问不存在的键值对，会增加容器的大小

利用这一特性，可以实现一个单词计数程序：

```
#include "test_include.h"
using namespace std;

void print(const map<string, int>::value_type &p) {
    cout << p.first << " occurs : " << p.second << " times" << endl;
}

int main(int argc, char **argv) {
    map<string, int> word_count;
    string word;

    while (cin >> word) {
        word_count[word]++;    //
```

```
    }

    for_each(word_count.begin(), word_count.end(), print);

}
```

在 map 中添加元素

刚才看到，采用下标的方式，可以给 map 添加元素，但更好的做法时采用 insert 插入一个 pair 对象。

```
#include "test_include.h"
using namespace std;

void print(const map<string, int>::value_type &p) {
    cout << p.first << " occurs : " << p.second << " times" << endl;
}

int main(int argc, char **argv) {
    map<string, int> word_count;
    string word;

    word_count.insert(map<string, int>::value_type("hello", 1));
    for_each(word_count.begin(), word_count.end(), print);
    cout << "--------------" << endl;
    word_count.insert(make_pair("test", 3));
    for_each(word_count.begin(), word_count.end(), print);
    cout << "----------------" << endl;

    pair<map<string, int>::iterator, bool> ret = word_count.insert(
            map<string, int>::value_type("hello", 4));
    for_each(word_count.begin(), word_count.end(), print);
    cout << "----------------" << endl;

    cout << ret.first->first << endl;      // hello
```

```
    cout << ret.first->second << endl;
    cout << ret.second << endl;       //

}
```

这里值得注意的是 insert 的返回值，返回了一个 pair 对象，第一个元素是指向该 key 的迭代器，第二个则表示插入是否成功

利用这个 insert，我们改写一些刚才的单词计数程序：

```
#include "test_include.h"
#include <stack>
#include <queue>
#include <typeinfo>
using namespace std;

void print(const map<string, int>::value_type &p) {
    cout << p.first << " occurs : " << p.second << " times" << endl;
}

int main(int argc, char **argv) {
    map<string, int> word_count;
    string word;

    while (cin >> word) {
        std::pair<std::map<std::string, int>::iterator, bool> ret =
                word_count.insert(
                        std::map<std::string, int>::value_type(word, 1));
        if(!ret.second)
        {
            ++ret.first->second;
        }
    }

    for_each(word_count.begin(), word_count.end(), print);

}
```

这里有必要解释++ret.first->second 这行语句

Ret 是个 pair 对象

Ret.first 表示插入元素的迭代器

Ret.first->second 表示该键值对中的 value，也就是单词出现的个数

## 在 map 中查找元素

刚才看到可以利用下标获取 value 的值，但是这样存在一个弊端，如果下标访问的是不存在的元素，那么会自动给 map 增加一个键值对，这显然不是我们所预期的。

采用 count 和 find 来解决问题

Count 仅仅能得出该元素是否存在，而 find 能够返回该元素的迭代器

```
#include "test_include.h"
#include <stack>
#include <queue>
#include <typeinfo>
using namespace std;

void print(const map<string, int>::value_type &p) {
    cout << p.first << " occurs : " << p.second << " times" << endl;
}

int main(int argc, char **argv) {
    map<string, int> word_count;
    string word;
    word_count["test"] = 10;
    word_count["foo"] = 5;
    word_count["bar"] = 12;
```

```
    cout << word_count.count("test") << endl;      //count 0 or 1

    cout << word_count.count("hello") << endl;

    map<string, int>::iterator iter = word_count.find("test");
    if (iter == word_count.end()) {
        cout << "not found" << endl;
    } else {
        cout << iter->first << " " << iter->second << endl;
    }

    for_each(word_count.begin(), word_count.end(), print);
    cout << "----------" << endl;
//  word_count.erase(iter);
    word_count.erase("test");
    for_each(word_count.begin(), word_count.end(), print);

}
```

# Set 容器

Set 类似于数学上的集合，仅仅表示某个元素在集合中是否存在，而不必关心它的具体位置。同样，set 中的元素互异，也就是无法两次插入相同的元素

使用方式和 map 类似，但是简单很多

```
#include "test_include.h"
using namespace std;



int main(int argc, char **argv) {
    set<int> s;

    for (size_t ix = 0; ix != 10; ++ix) {
```

```
            s.insert(ix);
            s.insert(ix);
        }

        cout << s.size() << endl; // 10

        s.insert(12);
        cout << s.size() << endl; // 11
        s.insert(12);
        cout << s.size() << endl;    //11

}


#include "test_include.h"
using namespace std;

void print(const string &s) {
        cout << s << " ";
}

int main(int argc, char **argv) {
        set<string> s;

        s.insert("hello");
        s.insert("world");
        s.insert("test");
        s.insert("foo");
        s.insert("bar");

        for_each(s.begin(), s.end(), print);
        cout << endl;

        s.erase("test");
        for_each(s.begin(), s.end(), print);
        cout << endl;

        set<string>::iterator iter = s.find("bar");
        if (iter == s.end()) {
                cout << "404" << endl;
        } else {
                s.erase(iter);
        }
        for_each(s.begin(), s.end(), print);
```

```
    cout << endl;

}
```

可以采用 set 实现一个停用词功能，在我们刚才的单词统计程序中，并不一定要统计所有的单词，而是要排除掉一部分

```
#include "test_include.h"
#include <fstream>
#include <stdexcept>
using namespace std;

ifstream &open_file(ifstream &is, const string &filename) {
    is.close();
    is.clear();
    is.open(filename.c_str());
    return is;
}

void restricted_wc(ifstream &input_file, const set<string> &exclude_words,
        map<string, int> &word_count) {
    string word;
    while (input_file >> word) {
        if (!exclude_words.count(word)) {
            ++word_count[word];
        }
    }
    input_file.close();
    input_file.clear();
}

void print(const string &s) {
    cout << s << " ";
}

void print_map(const map<string, int>::value_type &p) {
    cout << p.first << " occurs " << p.second << " times" << endl;
}

int main(int argc, char **argv) {
    set<string> exclude_words;
```

```
    map<string, int> word_count;

    exclude_words.insert("the");
    exclude_words.insert("a");
    exclude_words.insert("an");
    exclude_words.insert("I");
    exclude_words.insert("and");

    std::ifstream infile;
    string name = "in.txt";
    if (!open_file(infile, name)) {
        throw std::runtime_error("open file error!");
    }

    restricted_wc(infile, exclude_words, word_count);

    for_each(exclude_words.begin(), exclude_words.end(), print);
    cout << endl << "----------" << endl;

    for_each(word_count.begin(), word_count.end(), print_map);
    cout << endl;
}
```

## STL 算法

# OOP 之 复制控制

目录：



## 对象复制的时机：

根据一个类去显式或者隐式初始化一个对象
复制一个对象，将它作为实参传给一个函数
从函数返回时复制一个对象

那么如何完成对象复制的工作？这里需要的就是复制构造函数

## 复制构造函数

只有单个形参，而且该形参是本类类型对象的引用（常用 const 修饰），这样的构造函数成为复制控制函数

复制构造函数调用的时机就是在对象复制的时候。

如果什么也不做，编译器会自动帮我们合成一个默认的复制

构造函数，例如：

```
#include <iostream>
#include <string>

using namespace std;

class Student {
public:
	Student() :
		_id(0), _name("none"), _score(0) {
	}

	void debug() {
		cout << _id << " " << _name << " " << _score << endl;
	}


private:
	int _id;
	string _name;
	int _score;

};

int main(int argc, char **argv) {
	Student s1(23, "test", 77);
	Student s2;
	s1.debug();
	s2.debug();

}
```

观察程序运行的结果可以看到编译器提供的复制构造函数工作正常。

那么如果我们自己来定义复制构造函数，应该怎么写？

仍然是刚才的代码，我们加上：

```
#include <iostream>
#include <string>

using namespace std;

class Student {
public:
	Student() :
		_id(0), _name("none"), _score(0) {
	}
	Student(int id, const string &name, int score) :
		_id(id), _name(name), _score(score) {
	}


	void debug() {
		cout << _id << " " << _name << " " << _score << endl;
	}

	Student &operator=(const Student &rhs) {
		_id = rhs._id;
		_name = rhs._name;
		_score = rhs._score;
		return *this;
	}

private:
	int _id;
	string _name;
	int _score;

};

int main(int argc, char **argv) {
	Student s1(23, "test", 77);
	Student s2;
	s1.debug();
	s2.debug();
}
```

结果仍然是正确的。

现在来思考一个问题，既然编译器生成的复制构造函数工作

正常，那么什么时候需要我们自己来编写复制构造函数呢？

这就是下面的深拷贝和浅拷贝的问题。

## 深拷贝和浅拷贝

下面我们来实现一个简易的 String 类

头文件如下：

```
#ifndef STRING_H_
#define STRING_H_

#include <iostream>
#include <string.h>

namespace __str {

class string {
public:
	string();
	string(const char *);
	void debug();
	std::size_t size() const;
	~string();
private:
	char *_str;

};

} /* namespace __str */

#endif /* STRING_H_ */
```

Cpp 文件如下：

```
#include "_string.h"

namespace __str {

string::string() {
```

```
        _str = new char;
        _str[0] = 0;
}
string::string(const char *s) {
        _str = new char[strlen(s) + 1];
        strcpy(_str, s);
}

std::size_t string::size() const {
        return strlen(_str);
}

void string::debug() {
        std::cout << _str << std::endl;
}

string::~string() {
        delete[] _str;
}

} /* namespace __str */
```

看起来似乎正常，但是在 main 中运行程序就会崩溃，原因在哪里？

因为系统合成的复制构造函数，在复制 String 对象时，只是简单的复制其中的 str 的值，这样复制完毕后，就有两个 String 指向同一个内存区域，当对象析构时，发生两次 delete，导致程序错误

如何解决？

方案很简单，就是我们在复制 String 时，不去复制 str 的值，而是复制其指向的内存区域。

我们自定义复制构造函数如下：

```
string::string(const string &s) {
    _str = new char[s.size() + 1];
    strcpy(_str, s._str);
}
```

这两种复制对象的区别就是深拷贝和浅拷贝，定义如下：

深拷贝：

浅拷贝：

赋值运算符：

前面的复制构造函数说的是对象的复制，对象的赋值调用的则是对象的赋值运算符

这是我们首次接触到运算符重载，运算符重载是什么？

对于我们自定义的类（例如 Student），我们是无法进行比较操作的，因为我们自定义的类没有内置比较运算符（<= < > >= == !=），此时我们就可以通过运算符重载的规则给这些类加上运算符

这里我们需要重载的就是赋值运算符

例如 Student，如下：

```
#include <iostream>
#include <string>

using namespace std;
```

```
class Student {
public:
    Student() :
            _id(0), _name("none"), _score(0) {
    }
    Student(int id, const string &name, int score) :
            _id(id), _name(name), _score(score) {
    }


    void debug() {
        cout << _id << " " << _name << " " << _score << endl;
    }

    Student &operator=(const Student &rhs) {
        _id = rhs._id;
        _name = rhs._name;
        _score = rhs._score;
        return *this;
    }

private:
    int _id;
    string _name;
    int _score;

};

int main(int argc, char **argv) {
    Student s1(23, "test", 77);
    Student s2;
    s1.debug();
    s2.debug();

    s2 = s1;     // =

    s2.debug();

}
```

当然，如果我们什么也不做，系统也会自动合成一个赋值

运算符，但是什么时候需要我们自己来重载赋值运算符呢，仍然是考虑深拷贝和浅拷贝的问题

String 的赋值运算符如下：

```
string &string::operator =(const string &s) {
    if (this != &s) {
        delete[] _str;
        _str = new char[s.size() + 1];
        strcpy(_str, s._str);
    }
    return *this;
}
```

这里有两处值得注意的地方，一是赋值运算符应该检查自身赋值的情况（想想为什么），另一处是要返回自身的引用。

析构函数

对象销毁时调用的函数。一般用于释放动态分配的资源。

参考这个实例：

```
#include <iostream>
using namespace std;

class Test {
public:
    Test() {
        cout << "construct" << endl;
    }
    ~Test() {
        cout << "destroy" << endl;
    }
};

int main(int argc, char **argv) {
```

```
    //  Test t;
        {
    //      Test t;
            Test *p = new Test[10];
    //      Test &t2 = t;
            delete[] p;
        }

        cout << "test" << endl;
    }
```

对于 String 类则是这样编写的：

```
string::~string() {
    delete[] _str;
}
```

此时，String 类完整的源码如下：

```
#ifndef STRING_H_
#define STRING_H_

#include <iostream>
#include <string.h>

namespace __str {

class string {
public:
    string();
    string(const char *);
    string(const string &);
    void debug();
    std::size_t size() const;
    ~string();
    string &operator=(const string&);

private:
    char *_str;

};
```

```
} /* namespace __str */

#endif /* STRING_H_ */
```

CPP 文件如下：

```
#include "_string.h"

namespace __str {

string::string() {
	_str = new char;
	_str[0] = 0;
}
string::string(const char *s) {
	_str = new char[strlen(s) + 1];
	strcpy(_str, s);
}

string::string(const string &s) {
	_str = new char[s.size() + 1];
	strcpy(_str, s._str);
}

std::size_t string::size() const {
	return strlen(_str);
}

void string::debug() {
	std::cout << _str << std::endl;
}

string::~string() {
	delete[] _str;
}

string &string::operator =(const string &s) {
	if (this != &s) {
		delete[] _str;
		_str = new char[s.size() + 1];
		strcpy(_str, s._str);
	}
	return *this;
}
```

```
} /* namespace __str */
```

禁止复制：

如果想禁止复制一个类，应该怎么办？

显然需要把类的复制构造函数设为 private，但是这样以来类的 friend 仍然可以复制该类，于是我们只声明这个函数，而不去实现。

另外，如果你不需要复制该类的对象，最好把赋值运算也一并禁用掉。

所以这里的做法是：把复制构造函数和赋值运算符的声明设为 private 而不去实现。

后面更通用的做法是写一个类 noncopyable，凡是继承该类的任何类都无法复制和赋值

三法则：

如果类需要析构函数，那么它也需要复制构造函数和赋值运算符。

# OOP 之 运算符重载

## 目录：



## 运算符重载：

通过自定义运算符，程序员可以自定义类的操作。

运算符的重载有成员函数和友元两种形式。有的运算符可以选择任意一种实现，有的则必须使用友元函数的形式。

这里通过 String 类的编写，讲述常见的几个运算符的重载。

## 1.赋值运算符=

这个我们在复制控制一节写过，如下：

```
String &String::operator=(const String &s) {

    if (s != *this) {
        delete[] _str;
        _str = NULL;
```

```
        _str = new char[s.size() + 1];
        strcpy(_str, s.c_str());
    }
    return *this;
}
```

再次强调这里的三个要点：

1.要注意判断自身赋值的情况

2.要返回自身的引用

3.赋值运算符必须重载为成员函数的形式

这个运算符还可以接受 char *类型的参数

```
String &String::operator=(const char *str) {
    delete[] _str;
    _str = NULL;
    _str = new char[strlen(str) + 1];
    strcpy(_str, str);
    return *this;
}
```

2.复合赋值运算符+=

跟=类似，也必须重载为成员函数的形式

```
String &String::operator+=(const String &s) {

    char *tmp = new char[size() + s.size() + 1];
    strcpy(tmp, _str);
    strcat(tmp, s._str);

    delete[] _str;
    _str = NULL;

    _str = tmp;
    return *this;
}
String &String::operator+=(const char *s) {
    *this += String(s);
```

```
    return *this;
}
```

注意这里，重载的第二个+=运算符调用了第一个重载版本，这是避免重复代码的体现。

3.输出运算符 <<

输出运算符必须重载为友元函数的形式。因为<<运算符的第一个操作数必须为 IO 流

String 的输出运算符如下：

```
friend std::ostream &operator<<(std::ostream &os, const String &s);
```

这是类内部的友元声明

```
inline std::ostream &operator<<(std::ostream &os, const String &s) {
    os << s._str;
    return os;
}
```

注意输出运算符最后不要换行，把是否换行的权利交给使用类的用户

4.输入运算符>>

这个与输出运算符类似，但是参数不加 const，而且要注意 IO 失败的情况

类内部的声明：

```
friend std::istream &operator>>(std::istream &is, String &s);
```

函数实现：

```
inline std::istream &operator>>(std::istream &is, String &s) {
    char buf[1024];
    is >> buf;
    if (is) {
        s = buf;
    }
    return is;
}
```

5.加法运算符 +

加法运算符既可以重载为成员函数的形式，也可以重载为友元函数的形式。

但是重载为成员函数有个限制，我们知道标准库中的 string 可以实现下面三种加法：

A)string + char *
B)String + string
C)Char * + string

而采用成员函数的形式，限定了第一个操作符必须为 string 类型，所以我们只能实现上面两种运算。

所以最好的办法是采用友元函数重载的形式。

```
friend String operator+(const String &, const String &);
friend String operator+(const String &, const char *);
friend String operator+(const char *, const String &);
```

三个函数的实现如下：

```
inline String operator+(const String &lhs, const String &rhs) {
    String ret(lhs);
    ret += rhs;
    return ret;
}
```

```
inline String operator+(const String &lhs, const char *s) {
    return lhs + String(s);
}

inline String operator+(const char *s, const String &rhs) {
    return String(s) + rhs;
}
```

这里第一个+利用了已经重载的+=，而后面两个+运算符都利用了第一个加号。需要注意的是，加法运算法产生一个新的对象，参数代表的对象不会发生任何改变，所以返回的是一个局部变量

6.比较运算符 < <= >= > == !=

这些运算符都是对称形状的，所以最好采用友元函数的形式

String 的这些运算符如下：

```
friend bool operator==(const String &, const String &);
    friend bool operator!=(const String &, const String &);

    friend bool operator<(const String &, const String &);
    friend bool operator>(const String &, const String &);
    friend bool operator<=(const String &, const String &);
    friend bool operator>=(const String &, const String &);
```

实现如下：

```
inline bool operator==(const String &lhs, const String &rhs) {
    return strcmp(lhs._str, rhs._str) == 0;
}
inline bool operator!=(const String &lhs, const String &rhs) {
    return !(lhs == rhs);
}

inline bool operator<(const String &lhs, const String &rhs) {
    return strcmp(lhs._str, rhs._str) < 0;
}
```

```
inline bool operator>(const String &lhs, const String &rhs) {
    return strcmp(lhs._str, rhs._str) > 0;
}
inline bool operator<=(const String &lhs, const String &rhs) {
    return !(lhs > rhs);
}
inline bool operator>=(const String &lhs, const String &rhs) {
    return !(lhs < rhs);
}
```

7.下标操作符[]

下标运算符需要处理越界的情况，而且最后重载 const 和非 const 两个版本，返回值为引用，而不是局部变量（为什么？）

先看一个例子：

```
#include <vector>
#include <iostream>

using namespace std;

class Foo
{
public:
    int &operator[](std::size_t index)
    {
        return _data[index];
    }
    const int &operator[](std::size_t index) const
    {
        return _data[index];
    }

    void init()
    {
        for(size_t ix = 0; ix != 100; ++ix)
        {
            _data.push_back(ix);
        }
    }
```

```
private:
    vector<int> _data;
};

int main(int argc, char **argv) {

    Foo foo;
    foo.init();

    cout << foo[23] << endl;
}
```

String 类相应的实现如下：

```
char &String::operator[](std::size_t index) {
    return _str[index];
}
const char &String::operator[](std::size_t index) const {
    return _str[index];
}
```

通过实现 String 类总结运算符重载的原则：

A)= += 必须写成成员函数，而且返回值必须为自身引用

B)[]运算符必须为成员函数，而且有 const 和非 const 两个版本，分别可作为右值和左值

C)+ == != < <= > >=这些算术和关系运算符，最好定义为 friend 的形式

D)>> <<必须为 friend 形式，而且输入操作要处理错误的情况

E)有些操作要配合使用，例如重载了+，就最好重载+=，重载了==，就去重载!=

F)重载运算符应尽可能使用其他运算符的操作。比如!=使用==，+使

用了+=的操作

String 类的完整代码如下：

头文件 h

```
#ifndef STRING_H_
#define STRING_H_

#include <string.h>
#include <cstddef>
#include <iostream>

class String {
public:
    String();
    String(const char *);
    String(const String&);
    ~String();
    String &operator=(const String &);
    String &operator=(const char *);

    String &operator+=(const String &);
    String &operator+=(const char *);

    char &operator[](std::size_t index);
    const char &operator[](std::size_t index) const;

    std::size_t size() const;
    const char* c_str() const;
    void debug();

    friend String operator+(const String &, const String &);
    friend String operator+(const String &, const char *);
    friend String operator+(const char *, const String &);

    friend bool operator==(const String &, const String &);
    friend bool operator!=(const String &, const String &);

    friend bool operator<(const String &, const String &);
    friend bool operator>(const String &, const String &);
    friend bool operator<=(const String &, const String &);
```

```
    friend bool operator>=(const String &, const String &);

    friend std::ostream &operator<<(std::ostream &os, const String &s);
    friend std::istream &operator>>(std::istream &is, String &s);

private:
    char *_str;
};

inline String operator+(const String &lhs, const String &rhs) {
    String ret(lhs);
    ret += rhs;
    return ret;
}

inline String operator+(const String &lhs, const char *s) {
    return lhs + String(s);
}

inline String operator+(const char *s, const String &rhs) {
    return String(s) + rhs;
}
inline bool operator==(const String &lhs, const String &rhs) {
    return strcmp(lhs._str, rhs._str) == 0;
}
inline bool operator!=(const String &lhs, const String &rhs) {
    return !(lhs == rhs);
}

inline bool operator<(const String &lhs, const String &rhs) {
    return strcmp(lhs._str, rhs._str) < 0;
}
inline bool operator>(const String &lhs, const String &rhs) {
    return strcmp(lhs._str, rhs._str) > 0;
}
inline bool operator<=(const String &lhs, const String &rhs) {
    return !(lhs > rhs);
}
inline bool operator>=(const String &lhs, const String &rhs) {
    return !(lhs < rhs);
}

inline std::ostream &operator<<(std::ostream &os, const String &s) {
    os << s._str;
```

```
    return os;
}

inline std::istream &operator>>(std::istream &is, String &s) {
    char buf[1024];
    is >> buf;
    if (is) {
        s = buf;
    }
    return is;
}

#endif /* STRING_H_ */
```

## CPP 文件如下：

```
#include <iostream>
#include "String.h"

String::String() :
        _str(new char[1])         //""
{
    *_str = '\0';    // ""
}
String::String(const char *s) {
    _str = new char[strlen(s) + 1];
    strcpy(_str, s);
}
String::String(const String &s) {
    _str = new char[s.size() + 1];
    strcpy(_str, s.c_str());
}
String::~String() {
    delete[] _str;
    _str = NULL;
}
String &String::operator=(const String &s) {

    if (s != *this) {
        delete[] _str;
        _str = NULL;
        _str = new char[s.size() + 1];
        strcpy(_str, s.c_str());
```

```
    }
    return *this;
}

String &String::operator=(const char *str) {
    delete[] _str;
    _str = NULL;
    _str = new char[strlen(str) + 1];
    strcpy(_str, str);
    return *this;
}

String &String::operator+=(const String &s) {

    char *tmp = new char[size() + s.size() + 1];
    strcpy(tmp, _str);
    strcat(tmp, s._str);

    delete[] _str;
    _str = NULL;

    _str = tmp;
    return *this;
}
String &String::operator+=(const char *s) {
    *this += String(s);
    return *this;
}

char &String::operator[](std::size_t index) {
    return _str[index];
}
const char &String::operator[](std::size_t index) const {
    return _str[index];
}

std::size_t String::size() const {
    return strlen(_str);
}
const char* String::c_str() const {
    return _str;
}
void String::debug() {
    std::cout << _str << std::endl;
```

}

下面通过一个简易的智能指针的编写来展示如何重载成员操作运算符

代码如下：

```
#ifndef SMARTPTR_H_
#define SMARTPTR_H_

#include <string>
#include <iostream>

class Student {
public:
	Student(int id, const std::string &name) :
			_id(id), _name(name) {
		std::cout << "Create Student" << std::endl;
	}
	~Student() {
		std::cout << "Destroy Student" << std::endl;
	}

	void print() {
		std::cout << _id << " : " << _name << std::endl;
	}

private:
	int _id;
	std::string _name;
};

class SmartPtr {
public:
	SmartPtr();
	SmartPtr(Student *ptr);
```

```
    ~SmartPtr();

    void reset_ptr(Student *ptr);
    const Student *get_ptr() const;

    Student *operator->();
    const Student *operator->() const;

    Student &operator*();
    const Student &operator*() const;

private:
    Student *_ptr;

private:
    //prevent copy
    SmartPtr(const SmartPtr &);
    SmartPtr &operator=(const SmartPtr &);
};

#endif /* SMARTPTR_H_ */
```

CPP 文件如下：

```
#include "SmartPtr.h"

SmartPtr::SmartPtr() :
        _ptr(NULL) {

}
SmartPtr::SmartPtr(Student *ptr) :
        _ptr(ptr) {

}
SmartPtr::~SmartPtr() {
    delete _ptr;
}

void SmartPtr::reset_ptr(Student *ptr) {
    if (ptr != _ptr) {
        delete _ptr;
        _ptr = ptr;
    }
}
```

```
const Student *SmartPtr::get_ptr() const {
	return _ptr;
}

Student *SmartPtr::operator->() {
	return _ptr;
}
const Student *SmartPtr::operator->() const {
	return _ptr;
}

Student &SmartPtr::operator*() {
	return *_ptr;
}
const Student &SmartPtr::operator*() const {
	return *_ptr;
}
```

## 自增和自减运算符

```
#ifndef INTEGER_H_
#define INTEGER_H_

#include <iostream>

class Integer {
public:
	Integer(int num);
	~Integer();

	friend std::ostream &operator<<(std::ostream &os, const Integer &obj);
	Integer &operator++();
	Integer operator++(int);

private:
	int _num;
};

inline std::ostream &operator<<(std::ostream &os, const Integer &obj) {
	os << obj._num;
```

```
    return os;
}

#endif /* INTEGER_H_ */
```

## CPP 文件

```
#include "Integer.h"

Integer::Integer(int num) :
        _num(num) {

}
Integer::~Integer() {

}

Integer &Integer::operator++() {
    ++_num;
    return *this;
}

Integer Integer::operator++(int) {
    Integer ret(*this);
    ++(*this);
    return ret;
}
```

练习 Complex 类的编写：

实现一个复数类，可以进行简单的加减乘除操作，可以打印以及输出。

函数对象

假设我们有一个单词的容器，现在我们需要去统计里面长度在 6 个字符以上的单词的个数。

我们定义这样的一个函数：

```
 bool GT6(const string &s){
    return s.size() >= 6;
}
```

然后采用标准库的算法来统计单词数目如下：

```
vector<string>::size_type wc = std::count_if(words.begin(), words.end(), GT6);
```

这样做当然是对的，但是我们发现一个严重的问题，我们写的 GT6 仅能统计长度在 6 以上的单词，如果我需要统计 10 和 8，那么我还需要再次编写函数。

一种理想的情况是我们可以把比较的数目也当做参数传入。解决办法就是函数对象。

```
class GT {
public:
    GT(size_t val = 0) :
            bound_(val) {
    }
    bool operator() (const string &s){
        return s.size() >= bound_;
    }
private:
    std::string::size_type bound_;
};
```

此时我们就可以这样来调用：

```
vector<string>::size_type wc = std::count_if(words.begin(), words.end(),
            GT(6));
```

这样便很大程度上提高了灵活性。

# OOP 之继承

## --面向对象的第二个特征是：继承

## 类的派生

在编程的时候，经常遇到具有类似属性，但是细节或者行为存在差异的组件。在这种情形下，一种解决方案是将每个组件声明为一个类，并在每个类中实现所有的属性，这将导致大量重复的代码。另一种解决方案是使用继承，从同一个基类派生出类似的类，在基类中实现所有通用的功能，并在派生类中覆盖基本的功能，以实现让每个类都独一无二的行为。

C++派生语法如下：

```
//基类
class Base {
    // Base class members
};

//派生类
class Derived: public Base {
    // derived class members
};
```

前面我们写了一个 Person 类，如果我们需要编写 Student 和 Worker 类，必然也需要 Person 的三个属性：age/name/id，这时候我们采用继承来解决这个问题。

我们编写代码如下：

```
class Student: public Person {
```

```
private:
    string _school;

};

class Worker: public Person {
private:
    string _factory;

};
```

这里我们让 Student 和 Worker 继承了 Person，而且添加了各自的成员。继承之后的类具有两部分成员：一是从基类继承而来的成员，二是自己本身添加的成员。

我们在 main 中测试一下：

```
#include <iostream>
#include <string>
using namespace std;

class Person {
private:
    int _id;
    string _name;
    int _age;

public:

    Person() :
            _id(-1), _name("none"), _age(-1) {
    }

    Person(int id, const string &name, int age) :
            _id(id), _name(name), _age(age) {
    }

    void print(std::ostream &os) const {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }
};

class Student: public Person {
```

```
private:
    string _school;

};

class Worker: public Person {
private:
    string _factory;

};

int main() {

    Student s;
    s.print(cout);

}
```

可以看到 Student 的对象也可以正常调用 print 函数，说明 Student 类本身也具有这个函数。

## protected 关键字

上面的代码中，我们做一些改动，在 Student 中添加一个 test 方法，里面去改动 name 属性，如下：

```
class Student: public Person {
private:
    string _school;

public:
    void test(){
        _name = "test";
    }
};
```

编译报错：‘std::string Person::_name’ is private，这是因为 name 为 private 属性，只能在 Person 类内部访问，即使是派生类也无法访

问。

如果需要修改基类的某个属性，可以把基类的属性设为 protected。protected 关键字的含义是：该属性在本类和派生类的内部可以访问。

这里我们总结下三种访问权限：

假设有有 base 类和 derived 类（public 继承）：

private 成员作用域仅限于 base 内部

public 成员作用域在所有位置

protected 成员作用在 base 和 derived 内部

我们这里采用的继承方式为 public 继承，如果一个类里面有 public、private、protected 成员，他们经过 public 继承，在派生类中的访问标号为：public、不可访问、protected。

总结起来就是：

Private 为个人日记，在派生类中无法被访问

Protected 为家族秘籍，在派生类家族体系中可以访问

Public 为完全公开的东西，在派生类中可以随意的访问

Private 在派生类的内部不可见，但是通过基类本身的函数可以间接访问

## 继承体系下的函数调用

1.派生类调用从基类继承而来的函数

我们刚才见过，Student 可以正常调用继承来的 print 函数。

2.派生类自己额外添加的函数

我们为 Student 添加一个 test 函数。如下：

```
class Student: public Person {
private:
    string _school;

public:
    void test(){
        cout << "test" << endl;
    }
};
```

在 main 中调用 s.test() 仍然是正常的，实际上，对 test 的调用和继承体系无关。

3.派生类重写了从基类继承而来的函数

我们在 Student 中重写 print 函数。

不改变参数列表：

```
class Student: public Person {
private:
    string _school;

public:
    void test() {
        cout << "test" << endl;
    }
    void print(std::ostream &os) const {
        os << "print" << endl;
    }
};
```

打印结果为 print，说明调用的是我们自己定义的版本。

如果改变参数列表：

```
class Student: public Person {
private:
    string _school;
```

```
public:
    void test() {
        cout << "test" << endl;
    }
    void print() const {
        cout << "print" << endl;
    }

};
```

运行程序可知，此时仍然调用我们重写的版本。

如果两个函数都存在，如下：

```
class Student: public Person {
private:
    string _school;

public:
    void print() const {
        cout << "print1" << endl;
    }

    void print(std::ostream &os) const {
        os << "print2" << endl;
    }

};
```

在 main 中这样写：

```
Student s;
s.print();
s.print(cout);
```

打印结果为：

```
print1
print2
```

同样说明调用了我们重写的版本。这里无论哪种情况，只要我们在派生类中写了一个函数，和基类的函数重名（无论参数表是否相同），那么通过派生类对象调用的总是派生类重写的函数，我们称派生类的 print 隐藏了基类的 print。

那么基类的 print 就无法调用了吗？显然不是。我们对两个 print 做改动如下：

```
class Student: public Person {
private:
    string _school;

public:
    void print() const {
        cout << "print1" << endl;
        Person::print(cout);
    }

    void print(std::ostream &os) const {
        os << "print2" << endl;
        Person::print(os);
    }

};
```

打印结果为：

```
print1
id: -1 name: none age: -1
print2
id: -1 name: none age: -1
```

我们还可以在外面显式调用，在 main 中这样写：

```
s.Person::print(cout);
```

打印结果仍然是正常的。这就说明了被派生类隐藏的函数可以通过指定基类的类名来调用。

## 继承时的对象布局

派生类内部含有一个无名的基类对象，下面才是派生类自己的成

员，所以构造派生类时会先构造基类。

例如：

```
class Base {
public:
    Base() {
        cout << "Base create" << endl;
    }
};

class Derived: public Base {
public:
    Derived() {
        cout << "Derived create" << endl;
    }
};
```

在 main 中生成一个 Derived 对象，观察对象构造的顺序。

练习：自己写程序添加析构函数，观察析构的顺序。

## 派生类到基类的转化

根据上面的对象布局，派生类可以赋值给基类。例如：

```
Derived d;

Base b;

b = d;
```

这段代码是可行的，但是程序在运行的时候只复制了 derived 的基类部分，而对于派生类单独的部分被切除了，这种无意间裁剪数据导致 Derived 变成 Base 的行为成为切除。

要避免切除问题，最好的办法是传递 const 引用或者指针。

## 构造和析构顺序

前面提到派生类和基类的构造顺序，如果一个派生类还有其他类的对象做成员，那么基类构造函数、派生类构造函数和成员对象的构造函数，三者的执行顺序是怎样的？

```
class Base {
public:
    Base() {
        cout << "Base create" << endl;
    }

    ~Base() {
        cout << "Base destroy" << endl;
    }
};

class Other {
public:
    Other() {
        cout << "Other create" << endl;
    }
    ~Other() {
        cout << "Other destroy" << endl;
    }
};


class Derived: public Base {
public:
    Derived() {
        cout << "Derived create" << endl;
    }
    ~Derived() {
        cout << "Derived destroy" << endl;
    }
private:
    Other _other;
```

```
};
```

在 main 中生成 Derived 的对象，可以观测到他们的构造和析构顺序。

## 继承与构造函数

在前面的代码中，我们分别为 Person 提供了两个构造函数，分别是：

Person();

Person(int id, const string &name, int age);

现在我们想为 Student 提供两个构造函数，分别为：

Student(int id, const string name, int age, const string &school)和

Student();

第二个默认无参数的很容易，全部为空就可以了：

```
Student() :
        _school("none") {
}
```

但是第一个含有参数的构造函数，我们的目的是把前三个参数传给基类继承而来的三个属性。但是三个属性为 private，也就是说在 Student 中无法直接访问三个属性。

这里的解决方案是采用初始化列表，这样做：

```
Student(int id, const string name, int age, const string &school) :
        Person(id, name, age), _school(school) {

}
```

## 继承与复制控制

这里涉及到两个函数，拷贝构造函数和赋值运算符。手工实现这两者都必须显式调用基类的版本。

看下面的代码：

```
#include <iostream>
#include <string>
using namespace std;

class Person {
private:
    int _id;
    string _name;
    int _age;

public:

    Person() :
        _id(-1), _name("none"), _age(-1) {
    }

    Person(int id, const string &name, int age) :
        _id(id), _name(name), _age(age) {
    }

    void print(std::ostream &os) const {
        os << "id: " << _id << " name: " << _name << " age: " << _age <<
endl;
    }
};

class Student: public Person {
private:
    string _school;
public:
    Student() :
        _school("none") {
    }

    Student(int id, const string name, int age, const string &school) :
        Person(id, name, age), _school(school) {
```

```
    }

    Student(const Student &s) :
            _school(s._school) {

    }

    Student &operator=(const Student &s) {
        if (this != &s) {
            _school = s._school;
        }
        return *this;
    }

    void print(std::ostream &os) const {
        Person::print(os);
        os << _school << endl;
    }

};

class Worker: public Person {
private:
    string _factory;

};

int main() {

    Student s1(12, "zhangsna", 234, "test1");
    s1.print(cout);
    Student s2(s1);
    s2.print(cout);

    Student s3;
    s3 = s1;
    s3.print(cout);
}
```

最后的打印结果表明程序没有正常复制和赋值，因为我们在 Student 中没有显式调用 Person 的复制构造函数或者赋值运算符。

正确的做法是这样：

```
Student(const Student &s) :
        Person(s), _school(s._school) {

}

Student &operator=(const Student &s) {
    if (this != &s) {
        Person::operator=(s);
        _school = s._school;
    }
    return *this;
}
```

在复制构造函数中，我们在初始化列表中用 Student 对象 s 初始化基类，在赋值运算符中，则是显式调用了基类的赋值运算符。

## 禁止复制

在复制控制一节，我们学到禁止一个类复制的做法是将其拷贝构造函数和赋值运算符设为私有，而且只有声明，没有实现。

如果我们这里有 10 个类都需要禁止复制，那么可以每个类都进行上面的操作，但这样导致大量的重复代码，好的解决方案是采用继承。

我们写一个类：

```
class noncopyable {
public:
    noncopyable() {
    }
    ~noncopyable() {
    }
private:
    noncopyable(const noncopyable &);
    noncopyable &operator=(const noncopyable &);
};
```

这样凡是继承该类的类均失去了复制和赋值的能力。

# OOP 之动态绑定

--面向对象的第三个特征是：动态绑定

## 基类的指针或者引用指向派生类对象

看下面的程序：

```
Student s1(12, "zhangsna", 234, "test1");
Person *p = &s1;
```

完整代码：

```
#include <iostream>
#include <string>
using namespace std;

class Person {
private:
    int _id;
    string _name;
    int _age;

public:

    Person() :
            _id(-1), _name("none"), _age(-1) {
    }

    Person(int id, const string &name, int age) :
            _id(id), _name(name), _age(age) {
    }

    void print(std::ostream &os) const {
        os << "id: " << _id << " name: " << _name << " age: " << _age <<
endl;
    }
};

class Student: public Person {
private:
    string _school;
```

```
public:
    Student() :
            _school("none") {
    }

    Student(int id, const string name, int age, const string &school) :
            Person(id, name, age), _school(school) {

    }

    Student(const Student &s) :
            Person(s), _school(s._school) {

    }

    Student &operator=(const Student &s) {
        if (this != &s) {
            Person::operator=(s);
            _school = s._school;
        }
        return *this;
    }

    void print(std::ostream &os) const {
        Person::print(os);
        os << _school << endl;
    }

};

class Worker: public Person {
private:
    string _factory;

};

int main() {

    Student s1(12, "zhangsna", 234, "test1");
    Person *p = &s1;
}
```

编译通过，说明基类的指针可以指向派生类的对象。我们用这个

指针去调用程序，分下列几种情况：

1.调用基类中存在派生类没有改写的函数

我们把 Student 中的 print 注释掉。

```
Student s1(12, "zhangsna", 234, "test1");
Person *p = &s1;
p->print(cout);
```

打印结果，发现调用的是基类的版本

2.调用基类中不存在派生类单独添加的函数

我们在 Student 中单独添加一个函数 test：

```
void test(){
    cout << "test" << endl;
}
```

然后在 main 中用指针调用：

```
Student s1(12, "zhangsna", 234, "test1");
Person *p = &s1;
p->test();
```

发现编译失败，错误信息为：

error: ‘class Person’ has no member named ‘test’

3.调用派生类改写的函数（分参数相同和不同）

取消掉 print 的注释，然后添加另外一个无参数的 print。

```
void print(std::ostream &os) const {
    Person::print(os);
    os << _school << endl;
}

void print(){
    cout << "test" << endl;
}
```

运行程序，发现调用的依然是基类的版本

原因在于：我们可以让基类 Person 的指针指向 Student 对象，但

是这种情况下我们用指针 p 进行函数调用时，编译器以为指针 p 指向的其实是一个 Person 对象。所以上面的各种结果都有了合理的解释。

## .基类指针和派生类指针之间的相互转化

1.既然基类的指针可以指向派生类，那么派生类的指针能否转化为基类指针？

```
Student s1(12, "zhangsna", 234, "test1");
Student *ps =&s1;
Person *p = ps;
```

编译完全无问题，说明派生类指针可以自动转化为基类的指针，甚至不需要强制类型转换。

2.基类的指针能否转化为派生类指针？

```
Student s1(12, "zhangsna", 234, "test1");
Person *p = &s1;
Student *ps = p;
```

编译报错：error: invalid conversion from ‘Person*’ to ‘Student*’

加上强制类型转化：

```
Student *ps = (Student*)p;
```

此时编译通过。这种把基类指针转化为派生类指针的行为，需要进行强制类型转化，我们称其为“向下塑形”，向下塑形本质上是不安

全的，需要程序员人工保证安全性。

## 虚函数

前面我们用基类的指针指向派生类的对象，派生类改写了基类的 print 函数，结果指针调用 print，仍然是 Person 的版本，这是因为指针 p 把它指向的对象看作是一个 Person 对象。

我们现在做一些改动，在基类的 print 声明前加一个关键字 virtual，代码如下：

```
#include <iostream>
#include <string>
using namespace std;

class Person {
private:
    int _id;
    string _name;
    int _age;

public:

    Person() :
            _id(-1), _name("none"), _age(-1) {
    }

    Person(int id, const string &name, int age) :
            _id(id), _name(name), _age(age) {
    }

    virtual void print(std::ostream &os) const {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }
};

class Student: public Person {
```

```
    private:
        string _school;
    public:
        Student() :
                _school("none") {
        }

        Student(int id, const string name, int age, const string &school) :
                Person(id, name, age), _school(school) {

        }

        Student(const Student &s) :
                Person(s), _school(s._school) {

        }

        Student &operator=(const Student &s) {
            if (this != &s) {
                Person::operator=(s);
                _school = s._school;
            }
            return *this;
        }

        void print(std::ostream &os) const {
            Person::print(os);
            os << _school << endl;
        }

    };

    class Worker: public Person {
    private:
        string _factory;

    };

    int main() {

        Student s1(12, "zhangsna", 234, "test1");
        Person *p = &s1;
        p->print(cout);
    }
```

此时我们发现，print 调用的居然是 Student 的版本！

这种现象就叫做动态绑定，也就是所谓的多态。

用 virtual 修饰的函数就叫做虚函数。

## 静态绑定

回想函数重载部分，我们提过，构成一个函数唯一标示的要素有：函数名、形参列表，在类中还包括类名、const 属性，但是从不包括返回值。这些要素可以构成函数的唯一标示：函数签名。

事实上，对于我们以往的各种函数调用，编译器在编译期间就能根据调用的特征跟函数签名做对比，从而在编译器就能确定调用哪一个函数。

这种编译器间的绑定行为称为静态绑定，又称为静态联编。

## 动态绑定的定义和条件

我们前面提到了多态的情况，面向对象的多态就是动态绑定。这里需要两个条件：

1.调用的函数为虚函数

2.用基类的指针指向派生类的对象

满足了上面两个条件，编译器在碰到 p->print(cout)这类语句时，如果是以前的情况，那么编译器会根据 p 的类型是 Person*，print 是 Person

的成员函数而唯一确定其函数调用的版本。

但是在这里，因为满足了动态绑定的条件，编译器察觉到 print 是一个虚函数，所以编译器在编译时并不确定究竟调用哪个 print 版本，而是把这件事推迟到运行期，根据 p 实际指向的对象来决定调用哪个版本。

这种把函数调用的绑定行为推迟到运行期间的现象就叫做动态绑定。

总结如下：

当用 base 指针或引用操纵派生类对象时，如果调用的是虚函数，那么在运行期间根据指针指向的对象的实际类型来确定调用哪一个函数。如果不是虚函数，在编译期间，根据指针或引用的类型就可以确定调用哪一个函数。

注意：虚函数具有继承性，如果基类中具有某个 virtual 函数，派生类具有同名函数而且参数相同，返回值兼容基类版本，那么派生类的也是虚函数。

## 函数的覆盖和隐藏

有下列三种情况：

1.基类中有 virtual 函数 print，派生类中也有 print，参数相同。这

是动态绑定的情况。这叫做函数的覆盖。

2.基类中有 virtual 函数，派生类中也有同名函数，但是不满足动态绑定的情况。

3.基类中有非 vitual 函数，派生类具有同名函数，无论参数是否相同。

后两者称为函数的隐藏。

## 虚函数表

动态绑定的原理

虚指针

虚函数表

## 抽象类和纯虚函数

不能实例化的类称为抽象类。

## 虚析构函数

对于含有虚函数的继承体系，通常需要把基类的析构函数改为 virtual 函数。

如果没有声明为 virtual 会怎样？

例如 Person *p = &s;当执行 delete p 的时候，这里是静态绑定，所以调用的是 Person 的析构函数，Student 并没有被完整销毁。

如果我们声明为 virtual，那么根据动态绑定，p 指向的是 Student 对象，这么 delete 调用的是 Student 的析构函数。

练习：写程序验证：

```
#include <iostream>
using namespace std;
class Base {
public:
    ~Base() {
        cout << "Base destroy" << endl;
    }
};

class Derived: public Base {
public:
    ~Derived() {
        cout << "Derived destroy" << endl;
    }
};

int main() {

    Base *pb = new Derived;

    delete pb;
}
```

把 Base 函数设为 virtual，再次观察结果。

## 接口继承/实现继承

## 类的设计清单：

作者： 郭春阳 OOP 之动态绑定

1.类是否需要自己实现构造函数？

2.类的数据成员是否为 private？

3.每个构造函数是否初始化了所有的数据？

4.类是否需要析构函数？

5.类的析构函数是否需要为 virtual？

6.类是否需要拷贝构造函数？

7.类是否需要赋值运算符？

8.赋值运算符有没有正确处理自身赋值？

9.类是否需要一些相关联的操作符？

10.删除数组时是否调用了 delete[]？

11.在拷贝构造函数和赋值运算符的形参是否使用了 const？

12.当函数参数中有 reference 时，是否应该为 const？

13.是否适当的将类的成员函数声明为 const？

一个课堂练习：

当前有基类 Computer，它有一个纯虚函数 price 和 display，下面派生出许多品牌，那么某职员去采购不同品牌的电脑，如果把这不同品牌的电脑放入到一个容器中，从而方便的计算总价格？

# 08OOP 之泛型编程

多态的概念：

前面我们讲面向对象编程的时候，它的第三个特征为动态绑定。事实上，动态绑定属于运行期多态。前面我们讲过的函数重载属于编译期多态。

这里必须注意，传统的说法，OOP 的三大特征封装、继承、多态中的多态仅包含运行期多态。**编译期多态不是面向对象编程特征的一部分**。

下面我们来学习一种新的编译期多态-模板。它是泛型编程的重要组成部分。

## 函数模板

考虑一个最简单的求和问题，写成函数，可以有以下的版本：

```
int add(int a, int b){
    return a + b;
}

double add(double a, double b){

    return a + b;
}

string add(const string &a, const string &b){
    return a + b;
}
```

一个简单的求和程序，有没有办法写成一个统一的版本呢？方法

就是采用模板。

```
template <typename T>
T add(const T &a, const T &b){
    return a + b;
}
```

这就是我们写的第一个模板程序。我们观察这个程序，第一行的template <typename T>称为模板形参表，用来表示模板函数实际使用的类型或者值。

当我们使用函数模板的时候，编译器会推断哪个模板实参绑定到模板形参，一旦编译器确定了实际的模板形参，就称它为实例化。

对于 add(string(“abc”), string(“def”))的调用，编译器自动辨别出string 类型，使用它替代模板函数中的 T，并编译 string 版本的函数。

注意，调用模板函数时，参数必须完全匹配，例如，当调用 add(1, 3.4)时，会发生编译错误，原因在于编译器推断第一个类型为 int，第二个类型为 double，没有一个函数符合这个要求。

解决办法是可以写这样一个函数：

```
template <typename T1, typename T2>
T1 add(const T1 &a, const T2 &b){
    return a + b;
}
```

非类型模板形参

模板的形参未必都是类型，也可以是值。

例如：

```
template <typename T, size_t N>
```

```
void array_init(T (&parm)[N])
{
    for(size_t ix = 0; ix != N; ++ix){
        parm[ix] = 0;
    }
}
```

这是一个用来初始化数组的模板函数，它的第二个模板形参不是类型，而是一个数值。函数本身接收一个形参，该形参是数组的引用。当调用 array_init 的时候，编译器自动推导 T，并且计算出 N 的数值。

模板内部用 typename 指定类型

看下面的代码：

例如：

```
template <class Parm, class U>
Parm fcn(Parm *array, U value){
    Parm::size_type * p;
}
```

这段代码存在歧义，因为 Parm::size_type * p 既可以解释成定义一个变量，也可以解释成两个变量相乘。

解决方案是前面加一个 typename，来显式说明这里 Parm::size_type 是一个类型而不是变量。

编写模板程序的两条原则：

**1.模板的形参是 const 引用**

**2.函数体中的比较只使用<比较**

这样做的原因在于：使用 const 引用，模板参数就可以接受不可以复制的类型，而且采用引用参数，避免了对象的复制，可以加快程序

的效率。

只使用<的原因，可以让模板的用户更加清楚必须定义那些操作符。可以减少对类型的要求。

例如一个函数内同时具备>和<操作，如果我们直接用这两个运算符，那么用户必须为他们的类型同时重载两个操作符，但是只采用一个，就减少了用户的负担。

注意，**通过<实际上可以推断出所有的比较符**

例如 v1 < v2，那么 v1 > v2 可以看做 v2 < v1

v1 <= v2 可以看做 !(v1 > v2)

v1 >= v2 可以看做 !(v1 < v2)

练习：编写一个泛型的 swap 函数

## 类模板

我们在运算符重载一节，曾经编写了一个简单的智能指针类，下面我们把它改造成泛型的智能指针。

这里需要注意的是，我们要把类的定义实现全部放到同一个文件中，可以为 h 文件，但最好使用 hpp 文件。

```
#ifndef SMARTPTR_H_
#define SMARTPTR_H_

#include <string>
#include <iostream>

template <typename T>
class SmartPtr {
public:
	SmartPtr();
	explicit SmartPtr(T *ptr);
	~SmartPtr();

	void reset_ptr(T *ptr);
	const T *get_ptr() const;

	T *operator->();
	const T *operator->() const;

	T &operator*();
	const T &operator*() const;

private:
	T *ptr_;

private:
	//prevent copy
	SmartPtr(const SmartPtr &);
	SmartPtr &operator=(const SmartPtr &);
};




template <typename T>
inline SmartPtr<T>::SmartPtr() :
		ptr_(NULL) {

}

template <typename T>
inline SmartPtr<T>::SmartPtr(T *ptr) :
		ptr_(ptr) {
```

```
}

template <typename T>
inline SmartPtr<T>::~SmartPtr() {
	delete ptr_;
}

template <typename T>
inline void SmartPtr<T>::reset_ptr(T *ptr) {
	if (ptr != ptr_) {
		delete ptr_;
		ptr_ = ptr;
	}
}

template <typename T>
inline const T *SmartPtr<T>::get_ptr() const {
	return ptr_;
}

template <typename T>
inline T *SmartPtr<T>::operator->() {
	return ptr_;
}

template <typename T>
inline const T *SmartPtr<T>::operator->() const {
	return ptr_;
}

template <typename T>
inline T &SmartPtr<T>::operator*() {
	return *ptr_;
}

template <typename T>
inline const T &SmartPtr<T>::operator*() const {
	return *ptr_;
}

#endif /* SMARTPTR_H_ */
```

这里需要注意的是：

1.SmartPtr 不再是一个完整的类，正确使用方式是：SmartPtr<T>，所以我们以前接触的 STL，都是采用这种方式编写。

2.函数要声明为 inline。

3.每个函数前面都要加上模板参数。

课堂练习：

使用模板技术编写一个泛型队列。

# 网络编程之 TCP

下面将通过一种**问题驱动**的模式学习 TCP 网络编程。我们将通过编写一个 **echo 回射服务器**学习以下内容：

1.基本的 TCP 连接

2.readn、writen 以及 readline 函数的编写

3.使用 select 改进客户端

4.服务端处理 SIGPIPE 信号，客户端采用 shutdown 改进程序，close 和 shutdown 的区别

5.客户端采用 poll、epoll 改进程序

服务器并发的三种基本思路

1.采用 select、poll、epoll 编写服务器端

2.采用多进程编写服务器端

3.采用多线程编写服务器端

## 一、基本的 TCP 连接

在我们编写 TCP 程序之前，我们先回顾下 read 和 write 的用法。

```
ssize_t read(int fd, void *buf, size_t count);
```

read 的返回值有以下几种情况：

-1 表示读取错误，errno 置为相应的错误代码，如果是中断造成的 read 失败，errno 为 EINTR

0 表示读到了末尾，在 socket 中表示对方已经关闭了连接，这里接收到一个 FIN 请求，从而返回 0

>0 表示实际读到的字节数，这个返回值小于 n 并不是错误，因为 TCP 连接中可能并没有那么多的数据

ssize_t write(int fd, const void *buf, size_t count);

write 的返回值与 read 基本相同。

下面我们编写一个实际的 TCP 连接。

server 端：

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define ERR_EXIT(m) \
    do\
    {\
        perror(m);\
        exit(EXIT_FAILURE);\
    }while(0)

#define MAXLINE 1023



static void do_serviece(int peerfd) {
    char recvbuf[MAXLINE + 1];
```

```
        memset(recvbuf, 0x00, sizeof recvbuf);

        while (1) {
            int nread = read(peerfd, recvbuf, MAXLINE);
            if (nread < 0) {
                ERR_EXIT("read");
            }
            if (nread == 0) {
                fprintf(stdout, "peer close\n");
                break;
            }
            fprintf(stdout, "receive: %s", recvbuf);
            int nwrite = write(peerfd, recvbuf, nread);
            if (nwrite < 0) {
                ERR_EXIT("write");
            }
            memset(recvbuf, 0x00, sizeof recvbuf);
        }
    }

    int main(int argc, char **argv) {

        int listenfd = socket(AF_INET, SOCK_STREAM, 0);
        if (listenfd < 0) {
            ERR_EXIT("socket");
        }

        int on = 1;
        if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on))
    < 0)
            ERR_EXIT("setsockopt");

        struct sockaddr_in servaddr;
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(8989);
        servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
        socklen_t len = sizeof servaddr;
        int ret = bind(listenfd, (struct sockaddr*) &servaddr, len);
        if (ret < 0) {
            ERR_EXIT("bind");
        }

        ret = listen(listenfd, SOMAXCONN);
        if (ret < 0) {
```

```
            ERR_EXIT("listen");
        }

        struct sockaddr_in peeraddr;
        bzero(&peeraddr, sizeof peeraddr);
        len = sizeof peeraddr;
        int peerfd = accept(listenfd, (struct sockaddr*) &peeraddr, &len);
        if (peerfd < 0) {
            ERR_EXIT("accept");
        }
        fprintf(stdout, "IP= %s, port= %d\n", inet_ntoa(peeraddr.sin_addr),
                ntohs(peeraddr.sin_port));

        do_serviece(peerfd);

        close(peerfd);
        close(listenfd);

        return 0;
    }
```

client 端：

```
    #include <errno.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <unistd.h>

    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>

    #define ERR_EXIT(m) \
        do\
        {\
            perror(m);\
            exit(EXIT_FAILURE);\
        }while(0)

    #define MAXLINE 1023
```

```
static void do_service(int fd) {
    char recvbuf[MAXLINE + 1];
    char sendbuf[MAXLINE + 1];
    memset(recvbuf, 0x00, sizeof recvbuf);
    memset(sendbuf, 0x00, sizeof sendbuf);

    while (fgets(sendbuf, MAXLINE, stdin) != NULL) {

        int nwrite = write(fd, sendbuf, strlen(sendbuf));
        if (nwrite < 0) {
            ERR_EXIT("write");
        }
        int nread = read(fd, recvbuf, MAXLINE);
        if (nread < 0) {
            ERR_EXIT("read");
        }
        if (nread == 0) {
            fprintf(stdout, "peer close\n");
            break;
        }
        fprintf(stdout, "receive: %s", recvbuf);
        memset(recvbuf, 0x00, sizeof recvbuf);
        memset(sendbuf, 0x00, sizeof sendbuf);
    }
}

int main(int argc, char **argv) {

    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd < 0) {
        ERR_EXIT("socket");
    }

    struct sockaddr_in servaddr;
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(8989);
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    socklen_t len = sizeof servaddr;

    int ret = connect(fd, (struct sockaddr *) &servaddr, len);
    if (ret < 0) {
        ERR_EXIT("connect");
    }
```

```
        do_service(fd);

        close(fd);

        return 0;
    }
```

我们需要搞清楚一个问题：TCP 在传输数据的时，**什么数据需要处理大小端问题，什么数据不需要？**

这里要明白，UDP 是面向报文的协议，只要缓冲区足够，那么一次就可以接收到完整的报文。但是 TCP 是**面向字节流**的，发送的数据都是以字节为单位。

如果我们要发送字符串，“abc....xyz”，因为字符串是按照字节顺序拼接的，TCP 能够保证对方按照相同的顺序接收这些字节，所以对方接收到的也是”abc..xyz”。

但是数字就不一样，例如 int i = 0x12345678 ， 如果不考虑大小端的处理，假设 TCP 发送的顺序是 12 34 56 78，糟糕的是，接收方的字节序与发送方恰好相反，于是接收方收到数字为 0x78563412，从而出现了问题。

问题的根源在哪里？在于字符串在任何主机上都是顺序拼接，而 int 的 4 个字节在不同的主机上拼凑的方式不一样。

所以，**凡是以字节为单位的，不需要处理字节序的问题，以多个字节为基本单位的，就需要转化成网络字节序再发送**。

目前我们编写的 echo 程序存在这样一个问题：客户端关闭，服务器能及时感应到，也随之关闭，但是服务器关闭的时候，**客户端无法感知**。问题出在哪里？

原因： server 和 client 均有两个 fd，server 是阻塞在 fd 上，所以 client 关闭的时候，**server 的 read 函数迅速返回 0**，从而获知 client 已经关闭。但是 **client 是阻塞在 fgets 函数**上，所以 server 关闭的时候，client 无法知道，必须敲击回车，运行到 read 函数时才能获知 server 关闭。

## 二、readn、writen 以及 readline 函数的实现

TCP 存在一个所谓的**粘包**问题。

粘包问题是指：当发送数据过快时，例如快速发送两个报文，分别为 400、600 字节，此时我们接收方的缓冲区为 1024，那么我们可能接收到一个长度为 1000 的消息。我们即使知道这 1000 字节是由两条消息组成的，**也无法区分它们的边界**。此时就称两条报文粘合在了一起。处理的方式就是，我们可以在每条报文的前面加上要发送的字节数（**这个字节数要转化成网络序**），然后使用 writen 函数写入相应字节。接收方只需要先读取前面的字节数获取消息的长度为 n，然后**使用 readn 函数接收**即可。

readn 函数的返回值含义要与 read 相同。

实现如下：

```
ssize_t readn(int fd, void *buf, size_t count) {
    size_t nleft;
    ssize_t nread;          //BUG
    char *ptr;

    ptr = buf;
    nleft = count;

    while (nleft > 0) {
        nread = read(fd, ptr, nleft);
        if (nread < 0) {
            if (errno == EINTR) {
                nread = 0;
            } else {
                return -1;
            }
        } else if (nread == 0) {
            break;
        }
        nleft -= nread;
        ptr += nread;
    }
    return count - nleft;
}
```

writen 函数**必须返回写入的字节数，**否则就是错误。

writen 的实现如下：

```
ssize_t writen(int fd, const void *buf, size_t count) {
    size_t nleft;
    ssize_t nwritten;     //这里必须是有符号数!!!
    const char *ptr;

    ptr = buf;
    nleft = count;
```

```
    while (nleft > 0) {
        nwritten = write(fd, ptr, nleft);
        if (nwritten < 0) {
            if (errno == EINTR) {
                //nwritten = 0;
                continue;
            } else {
                return -1;
            }
        } else if (nwritten == 0) {
            continue;
        }
        nleft -= nwritten;
        ptr += nwritten;
    }

    return count;
}
```

此时我们用 readn 和 writen 改进我们的程序如下：

需要添加一个结构体

```
#define MAXLINE 1023

struct pack {
    int len;
    char data[MAXLINE + 1];
};
```

server 端：

```
static void do_service(int peerfd) {
//  char recvbuf[MAXLINE + 1];
//  memset(recvbuf, 0x00, sizeof recvbuf);

    struct pack recvpac;
    memset(&recvpac, 0x00, sizeof recvpac);

    while (1) {
        int nread = readn(peerfd, &recvpac.len, 4);
        if (nread < 0) {
```

```
            ERR_EXIT("nread");
        }
        if (nread < 4) {
            fprintf(stdout, "peer close\n");
            break;
        }
        int nlen = ntohl(recvpac.len);
        fprintf(stdout, "len: %d\n", nlen);

        nread = readn(peerfd, recvpac.data, nlen);
        if (nread < 0) {
            ERR_EXIT("nread");
        }
        if (nread < nlen) {
            fprintf(stdout, "peer close\n");
            break;
        }
        fprintf(stdout, "receive: %s", recvpac.data);


        writen(peerfd, &recvpac, 4 + nlen);

        memset(&recvpac, 0x00, sizeof recvpac);
    }
}
```

client 端：

```
static void do_service(int fd) {

    struct pack recvbuf;
    struct pack sendbuf;
    memset(&recvbuf, 0x00, sizeof recvbuf);
    memset(&sendbuf, 0x00, sizeof sendbuf);

    while (fgets(sendbuf.data, MAXLINE, stdin) != NULL) {

        int nlen = strlen(sendbuf.data);
        sendbuf.len = htonl(nlen);
        writen(fd, &sendbuf, nlen + sizeof (int));

        int nread = readn(fd, &recvbuf.len, 4);
        if(nread < 0){
```

```
                ERR_EXIT("nread");
            }
            if(nread < 4){
                fprintf(stdout, "peer close\n");
                break;
            }

            fprintf(stdout, "len: %d\n", nlen);
            nread = readn(fd, recvbuf.data, nlen);
            if (nread < 0) {
                ERR_EXIT("nread");
            }
            if (nread == 0) {
                fprintf(stdout, "peer close\n");
                break;
            }
            fprintf(stdout, "receive: %s", recvbuf.data);
            memset(&recvbuf, 0x00, sizeof recvbuf);
            memset(&sendbuf, 0x00, sizeof sendbuf);
        }
    }
```

我们的粘包问题，有了初步的解决方案。但是在现实中的很多应用层协议，在发送消息的时候是**以\r\n 作为一条消息边界**的，例如HTTP，而不是采用固定的字节数。

所以我们可以编写一个 readline 函数，每次读取消息，直到遇见\n 为止。

那么该如何实现这个函数呢？

我们平时读取消息后，数据立刻从缓冲区清除，但是 TCP 中有一个函数 recv，它的声明如下：

ssize_t recv(int sockfd, void *buf, size_t len, int flags);

最后一个参数是标志位，如果我们设置为 MSG_PEEK，那么在调用 recv 的时候，数据读取到缓冲区，但是**缓冲区中并不擦除数据**。

这里可以把这个 recv 函数调用，**看做一个斥候，为我们后续的数据读取探路**。

所以这里我们给出 readline 函数的实现，我们需要编写另外一个辅助函数 recv_peek：

```
ssize_t recv_peek(int sockfd, void *buf, size_t len) {
    int nread;
    while (1) {
        nread = recv(sockfd, buf, len, MSG_PEEK);
        if (nread < 0 && errno == EINTR) {    //被中断则继续读取
            continue;
        }
        if (nread < 0) {
            return -1;
        }
        break;
    }
    return nread;
}

ssize_t readline(int sockfd, void *buf, size_t maxline) {
    int nread;     //一次 IO 读取的数量
    int nleft;//还剩余的字节数
    char *ptr;     //存放数据的指针的位置
    int ret;   //readn 的返回值
    int total = 0; //目前总共读取的字节数

    nleft = maxline;
    ptr = buf;

    while (nleft > 0) {
        ret = recv_peek(sockfd, ptr, nleft);
        //注意这里读取的字节不够，绝对不是错误!!!
        if (ret <= 0) {
            return ret;
        }

        nread = ret;
        int i;
        for (i = 0; i < nread; ++i) {
            if (ptr[i] == '\n') {
```

```
                ret = readn(sockfd, ptr, i + 1);
                if (ret != i + 1) {
                    return -1;
                }
                total += ret;
                return total;  //返回此行的长度  '\n'包含在其中
            }
        }

        ret = readn(sockfd, ptr, nread);
        if (ret != nread) {
            return -1;
        }
        nleft -= nread;
        total += nread;
        ptr += nread;
    }
    return maxline;
}
```

所以我们此时再次改进程序，每次发送数据使用 writen，**因为我们用 fgets 获取数据，所以数据末尾总是\n，**接收方采用 readline 就可以了。

我们再次改进 echo 回射服务器的核心代码：

```
static void do_service(int fd) {
    char recvbuf[MAXLINE + 1];
    memset(recvbuf, 0x00, sizeof recvbuf);

    while (1) {
        int ret = readline(fd, recvbuf, MAXLINE);
        if (ret == -1) {
            ERR_EXIT("readline");
        }
        if(ret == 0){
            fputs("peer close", stdout);
            break;
        }
//      fputs(recvbuf, stdout);
        fprintf(stdout, "receive: %s", recvbuf);
```

```
            writen(fd, recvbuf, strlen(recvbuf));
            memset(recvbuf, 0x00, sizeof recvbuf);
        }
    }
```

客户端代码如下：

```
    static void do_service(int fd) {
        char recvbuf[MAXLINE + 1];
        char sendbuf[MAXLINE + 1];
        memset(recvbuf, 0x00, sizeof recvbuf);
        memset(sendbuf, 0x00, sizeof sendbuf);

        while (fgets(sendbuf, MAXLINE, stdin) != NULL) {
            writen(fd, sendbuf, strlen(sendbuf));
            int ret = readline(fd, recvbuf, MAXLINE);
            if (ret == -1) {
                ERR_EXIT("readline");
            }
            if (ret == 0) {
                fputs("peer close", stdout);
                break;
            }
            fprintf(stdout, "receive: %s", recvbuf);
            memset(recvbuf, 0x00, sizeof recvbuf);
            memset(sendbuf, 0x00, sizeof sendbuf);
        }

    }
```

## 三、用 select 改进客户端程序

回顾我们之前遇到的问题：客户端阻塞在 fgets 上，导致不能及时接收到 server 发来的 FIN 请求，下面我们尝试着用 select 解决这个问题。

我们目前遇到的难题在于 client 只能阻塞在一个 fd 上，如果阻塞在标准输入上，那么服务器关闭我们无法感知，如果阻塞在 read（或者 readn、readline）函数上，那么我们无法及时输入。

**打一个比方**：老师给学生布置了课上习题，然后要检查他们做的正确与否。我们目前的程序是这样的，老师走到学生 A 处，等待他做完，然后检查，再去找学生 B，**如果某个学生在那里消耗时间，老师和后面的同学也只能耐心等待**。

这样显然不行，稍微聪明点的人就知道应该这样做：**老师在台上等待，谁做完谁举手，然后老师过去检查**，这样就很大程度上提高了效率。

这就是我们所要讲述的 IO 复用模型。

select 用法参考：

http://www.cnblogs.com/Anker/archive/2013/08/14/3258674.html

这里我们采用 select 模型来改进我们的客户端。

```
void do_service(int fd) {
    char recvbuf[MAXLINE + 1];
    char sendbuf[MAXLINE + 1];
    memset(recvbuf, 0x00, sizeof recvbuf);
    memset(sendbuf, 0x00, sizeof sendbuf);

    fd_set rset;
    int maxfd;
    int nready;
    int stdin_fd = fileno(stdin);
    maxfd = (stdin_fd > fd) ? stdin_fd : fd;
    FD_ZERO(&rset);
    int stdin_eof = 0;

    while (1) {
        if (stdin_eof == 0) {
            FD_SET(stdin_fd, &rset);
        }
        FD_SET(fd, &rset);
```

```
            nready = select(maxfd + 1, &rset, NULL, NULL, NULL);
            if (nready < 0) {
                ERR_EXIT("select");
            }
            if (nready == 0) {
                continue;
            }
            if (FD_ISSET(stdin_fd, &rset)) {
                if (fgets(sendbuf, MAXLINE, stdin) == NULL) {
                    //break;
                    close(fd);
                    sleep(4);
                    exit(EXIT_FAILURE);


                } else {
                    if(strcmp(sendbuf, "\n") == 0){
                        continue;
                    }
                    writen(fd, sendbuf, strlen(sendbuf));
                    memset(sendbuf, 0x00, sizeof sendbuf);
                }
            }
            if (FD_ISSET(fd, &rset)) {
                int ret = readline(fd, recvbuf, MAXLINE);
                if (ret == -1) {
                    ERR_EXIT("readline");
                }
                if (ret == 0) {
                    fputs("server close\n", stdout);
                    break;
                }
                fprintf(stdout, "receive: %s", recvbuf);
                memset(recvbuf, 0x00, sizeof recvbuf);
            }
        }

    }
```

## 四、SIGPIPE 信号以及 shutdown 与 close 的区别

OK，现在我们的客户端可以及时获取 server 关闭的消息了。

**我们进行一些破坏活动。**

**server 的 read 和 writen 之间加入一个 sleep(4)，代表处理数据**

依次启动 server/client，在 client 中迅速发送两个字符串，然后 Ctrl+D，然后你就看到，回射的消息没有收到，服务器崩溃。

让我们分析其中的原因：

1.client 发送两条数据，然后 ctrl+D，这时 client 调用 close **同时关闭了读写方向**。所以 client 发送了一个 **FIN** 请求。

2.server 收到第一条信息，然后睡眠，4s 后，发送数据，因为此时对方已经无法接收，所以 server **收到一个 RST 报文**。

3.server 收到第二条消息，4s 后再次发送，此时**往已经接收到 RST 的连接中发送数据，server 端收到一个 SIGPIPE 信号**，从而崩溃。

原因分析清楚了，如何解决呢？

客户端可以**改用 shutdown**。这里先解释一下 close 和 shutdown 的区别，有两点：

1.close 同时关闭读和写两个方向，而 shutdown 可以选择只关闭其中一个方向。

2.close 仅当某 fd 的引用计数为 0 时才真正关闭，而 shutdown 是立刻关闭。

这里我们主要利用第一点，client 按下 Ctrl+D 后，只关闭发送端，接收端依然开启，所以 server 不会遇到 RST 请求以及 SIGPIPE 信号。

代码如下：

```
//

    static void do_service(int fd) {
        char recvbuf[MAXLINE + 1];
        char sendbuf[MAXLINE + 1];
        memset(recvbuf, 0x00, sizeof recvbuf);
        memset(sendbuf, 0x00, sizeof sendbuf);

        fd_set rset;
        int maxfd;
        int nready;
        int stdin_fd = fileno(stdin);
        maxfd = (stdin_fd > fd) ? stdin_fd : fd;
        FD_ZERO(&rset);
        int stdin_eof = 0;

        while (1) {
            if (stdin_eof == 0) {
                FD_SET(stdin_fd, &rset);
            }
            FD_SET(fd, &rset);
            nready = select(maxfd + 1, &rset, NULL, NULL, NULL);
            if (nready < 0) {
                ERR_EXIT("select");
            }
            if (nready == 0) {
                continue;
            }
            if (FD_ISSET(stdin_fd, &rset)) {
                if (fgets(sendbuf, MAXLINE, stdin) == NULL) {
                    //break;
                    /*close(fd);
                    sleep(4);
                    exit(EXIT_FAILURE); */
                    stdin_eof = 1; //停止监听该描述符
                    shutdown(fd, SHUT_WR);

                } else {
                    if(strcmp(sendbuf, "\n") == 0){
                        continue;
                    }
                    writen(fd, sendbuf, strlen(sendbuf));
```

```
                    memset(sendbuf, 0x00, sizeof sendbuf);
                }
            }
            if (FD_ISSET(fd, &rset)) {
                int ret = readline(fd, recvbuf, MAXLINE);
                if (ret == -1) {
                    ERR_EXIT("readline");
                }
                if (ret == 0) {
                    fputs("server close\n", stdout);
                    break;
                }
                fprintf(stdout, "receive: %s", recvbuf);
                memset(recvbuf, 0x00, sizeof recvbuf);
            }
        }

    }
```

我们似乎解决了问题，但是这种**依靠 client 端修改代码的方式并不稳妥！**服务器不能把自身的安全交给客户端来保证。

所以稳妥的方式是 **server 端必须处理 SIGPIPE 信号**。

如下：

```
signal(SIGPIPE, SIG_IGN);
```

## 五、用 poll 改进客户端

前面我们使用 select 改进了客户端，但是 select 存在一系列的缺点，

比如：

于是我们采用另一种模型：poll

poll 的用法参见：

http://www.cnblogs.com/Anker/archive/2013/08/15/3261006.html

使用 poll 编写的客户端程序如下：

```
static void do_service(int fd) {
    char recvbuf[MAXLINE + 1];
    char sendbuf[MAXLINE + 1];
    memset(recvbuf, 0x00, sizeof recvbuf);
    memset(sendbuf, 0x00, sizeof sendbuf);


    int maxi = 0;//最大可用的下标
    struct pollfd client[2];
    int nready;
    client[0].fd = fd;
    client[0].events = POLLIN;
    int stdin_fd = fileno(stdin);
    client[1].fd = stdin_fd;
    client[1].events = POLLIN;
    maxi = 1;

    while (1) {
        nready = poll(client, maxi + 1, -1);
        if (nready < 0) {
            if (errno == EINTR) {
                continue;
            }
            ERR_EXIT("poll");
        }
        if (nready == 0) {
            continue;
        }

        //if (FD_ISSET(stdin_fd, &rset)) {
        if (client[1].revents & POLLIN) {
            if (fgets(sendbuf, MAXLINE, stdin) == NULL) {
                //break;
                //stdin_eof = 1; //停止监听该描述符
                client[1].fd = -1; //停止监听
                shutdown(fd, SHUT_WR);

            } else {
                if (strcmp(sendbuf, "\n") == 0) {
                    continue;
                }
                writen(fd, sendbuf, strlen(sendbuf));
                memset(sendbuf, 0x00, sizeof sendbuf);
            }
```

```
            }
            //if (FD_ISSET(fd, &rset)) {
            if (client[0].revents & POLLIN) {
                int ret = readline(fd, recvbuf, MAXLINE);
                if (ret == -1) {
                    ERR_EXIT("readline");
                }
                if (ret == 0) {
                    fputs("server close\n", stdout);
                    client[0].fd = -1;
                    break;
                }
                fprintf(stdout, "receive: %s", recvbuf);
                memset(recvbuf, 0x00, sizeof recvbuf);
            }
        }

    }
```

## 六、用 epoll 改进客户端

select 和 poll 都存在这样一些缺点：

epoll 用法如下：

http://www.cnblogs.com/Anker/archive/2013/08/17/3263780.html

用 epoll 改写客户端如下：

```
static void do_service(int fd) {
    char recvbuf[MAXLINE + 1];
    char sendbuf[MAXLINE + 1];
    memset(recvbuf, 0x00, sizeof recvbuf);
    memset(sendbuf, 0x00, sizeof sendbuf);

    int epollfd = epoll_create(2);
    if (epollfd == -1) {
```

```
        ERR_EXIT("epoll_create");
    }
    struct epoll_event events[2];
    struct epoll_event ev;
    ev.data.fd = fd;
    ev.events = EPOLLIN;
    int ret = epoll_ctl(epollfd, EPOLL_CTL_ADD, fd, &ev);
    if (ret == -1) {
        ERR_EXIT("epoll_ctl");
    }
    int stdin_fd = fileno(stdin);
    ev.data.fd = stdin_fd;
    ev.events = EPOLLIN;
    ret = epoll_ctl(epollfd, EPOLL_CTL_ADD, stdin_fd, &ev);
    if (ret == -1) {
        ERR_EXIT("epoll_ctl");
    }
    int nready;

    while (1) {

        //nready = poll(client, maxi + 1, -1);
        nready = epoll_wait(epollfd, events, 2, -1);
        if (nready < 0) {
            if (errno == EINTR) {
                continue;
            }
            ERR_EXIT("epoll_wait");
        }
        if (nready == 0) {
            continue;
        }

        int i;
        for (i = 0; i < nready; ++i) {

            //if (client[1].revents & POLLIN) {
            if (events[i].data.fd == stdin_fd) {

                if (fgets(sendbuf, MAXLINE, stdin) == NULL) {
                    //break;
                    //stdin_eof = 1; //停止监听该描述符
                    //client[1].fd = -1; //停止监听
                    ev.data.fd = stdin_fd;
```

```
                    epoll_ctl(epollfd, EPOLL_CTL_DEL, stdin_fd, &ev); // 从
events 中删除该 fd
                    shutdown(fd, SHUT_WR);

                } else {
                    if (strcmp(sendbuf, "\n") == 0) {
                        continue;
                    }
                    writen(fd, sendbuf, strlen(sendbuf));
                    memset(sendbuf, 0x00, sizeof sendbuf);
                }
            }

            //if (client[0].revents & POLLIN) {
            if(events[i].data.fd == fd){
                int ret = readline(fd, recvbuf, MAXLINE);
                if (ret == -1) {
                    ERR_EXIT("readline");
                }
                if (ret == 0) {
                    fputs("server close\n", stdout);
                    //client[0].fd = -1;
                    ev.data.fd = fd;
                    epoll_ctl(epollfd, EPOLL_CTL_DEL, fd, &ev); //delete
                    close(epollfd);
                    exit(EXIT_SUCCESS);  //直接退出程序
                }
                fprintf(stdout, "receive: %s", recvbuf);
                memset(recvbuf, 0x00, sizeof recvbuf);
            }

        }
    }

}
```

在我们改进客户端的过程中，select、poll、epoll 的大概模型是相似的。

大概的流程都是这样的：

1.添加监听的描述符和时间

2.监听，等待返回

3.处理读事件

4.处理写事件

5.继续下次循环

三种并发思路

1.IO 多路复用

2.多进程

3.多线程

多线程服务器端的基本框架

MutexLock 和 Condition

Thread　ThreadPool

InetAddress　Socket

Select/Poller/Epoller

TcpServer TcpClient

IO 多路复用

select 编写服务器

```
int main(int argc, char **argv) {

    signal(SIGPIPE, SIG_IGN);
    int listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd < 0) {
        ERR_EXIT("socket");
    }

    int on = 1;
    if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on))
< 0)
        ERR_EXIT("setsockopt");
```

```
struct sockaddr_in servaddr;
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(8989);
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
socklen_t len = sizeof servaddr;
int ret = bind(listenfd, (struct sockaddr*) &servaddr, len);
if (ret < 0) {
    ERR_EXIT("bind");
}

ret = listen(listenfd, SOMAXCONN);
if (ret < 0) {
    ERR_EXIT("listen");
}

int i;
int client[FD_SETSIZE];
for (i = 0; i < FD_SETSIZE; ++i) {
    client[i] = -1;
}
int maxi = 0;
int maxfd = listenfd;
int nready;
fd_set allset;
fd_set rset;
FD_ZERO(&allset);
FD_ZERO(&rset);
FD_SET(listenfd, &allset);

while (1) {
    rset = allset;
    nready = select(maxfd + 1, &rset, NULL, NULL, NULL);
    if (nready == -1) {
        if (errno == EINTR) {
            continue;
        } else {
            ERR_EXIT("select");
        }
    }
    if (nready == 0) {
        continue;
    }
    if (FD_ISSET(listenfd, &rset)) {
```

```
        struct sockaddr_in peeraddr;
        bzero(&peeraddr, sizeof peeraddr);
        len = sizeof peeraddr;

        //accept
        int peerfd = accept(listenfd, (struct sockaddr*) &peeraddr, &len);
        if (peerfd == -1) {
            ERR_EXIT("accept");
        }
        //加入 clients
        int i;
        for (i = 0; i < FD_SETSIZE; ++i) {
            if (client[i] == -1) {
                client[i] = peerfd;
                if (i > maxi) {
                    maxi = i;
                }
                break;
            }
        }

        //too many
        if (i == FD_SETSIZE) {
            fprintf(stderr, "too many clients\n");
            exit(EXIT_FAILURE);
        }

        //加入 allset
        FD_SET(peerfd, &allset);
        if (peerfd > maxfd) {
            maxfd = peerfd;
        }
        fprintf(stdout, "IP = %s, port = %d\n",
                inet_ntoa(peeraddr.sin_addr), ntohs(peeraddr.sin_port));

        //如果等于零，说明其他 fd 不需要操作
        if (--nready <= 0) {
            continue;
        }

    }

    int i;
    for (i = 0; i <= maxi; ++i) {
```

```
                int peerfd = client[i];
                if (peerfd == -1) {
                    continue;
                }
                if (FD_ISSET(peerfd, &rset)) {
                    char recvbuf[MAXLINE + 1] = { 0 };
                    int ret = readline(peerfd, recvbuf, MAXLINE);
                    if (ret == -1) {
                        ERR_EXIT("readline");
                    }
                    if (ret == 0) {
                        fputs("client close\n", stdout);
                        FD_CLR(peerfd, &allset);
                        client[i] = -1;
                        close(peerfd);
                        continue;
                    }
                    fprintf(stdout, "receive: %s", recvbuf);
                    //sleep(4);
                    writen(peerfd, recvbuf, strlen(recvbuf));
                    //write(peerfd, "test\n", strlen("test\n"));

                    if (--nready <= 0) {
                        break;
                    }

                }

            }

        }

    //  close(peerfd);
        close(listenfd);

        return 0;
    }
```

poll 编写服务器

```
    int main(int argc, char **argv) {
```

```
        signal(SIGPIPE, SIG_IGN);
        int listenfd = socket(AF_INET, SOCK_STREAM, 0);
        if (listenfd < 0) {
            ERR_EXIT("socket");
        }

        int on = 1;
        if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on))
< 0)
            ERR_EXIT("setsockopt");

        struct sockaddr_in servaddr;
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(8989);
        servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
        socklen_t len = sizeof servaddr;
        int ret = bind(listenfd, (struct sockaddr*) &servaddr, len);
        if (ret < 0) {
            ERR_EXIT("bind");
        }

        ret = listen(listenfd, SOMAXCONN);
        if (ret < 0) {
            ERR_EXIT("listen");
        }


        struct pollfd client[2048];
        int i;
        for (i = 0; i < 2048; ++i) {
            client[i].fd = -1;
        }
        client[0].fd = listenfd;
        client[0].events = POLLIN;
        int maxi = 0;
        int nready;

        while (1) {
            /*rset = allset;
              nready = select(maxfd + 1, &rset, NULL, NULL, NULL); */
            nready = poll(client, maxi + 1, -1);
            if (nready == -1) {
                if (errno == EINTR) {
```

```
            continue;
        } else {
            ERR_EXIT("poll");
        }
    }
    if (nready == 0) {
        continue;
    }
    //if (FD_ISSET(listenfd, &rset)) {
    if (client[0].revents & POLLIN) {
        struct sockaddr_in peeraddr;
        bzero(&peeraddr, sizeof peeraddr);
        len = sizeof peeraddr;

        //accept
        int peerfd = accept(listenfd, (struct sockaddr*) &peeraddr, &len);
        if (peerfd == -1) {
            ERR_EXIT("accept");
        }


        int i;
        for (i = 0; i < 2048; ++i) {
            if (client[i].fd == -1) {
                client[i].fd = peerfd;
                client[i].events = POLLIN;    //容易遗漏
                if (i > maxi) {
                    maxi = i;
                }
                break;
            }
        }

        if (i == 2048) {
            fprintf(stderr, "too many clients\n");
            exit(EXIT_FAILURE);
        }

        fprintf(stdout, "IP = %s, port = %d\n",
                inet_ntoa(peeraddr.sin_addr), ntohs(peeraddr.sin_port));

        //如果等于零，说明其他 fd 不需要操作
        if (--nready <= 0) {
            continue;
```

```
            }

        }

        int i;
        //for (i = 0; i <= maxi; ++i) {
        for (i = 1; i <= maxi; ++i) {
            int peerfd = client[i].fd;
            if (peerfd == -1) {
                continue;
            }
            //if (FD_ISSET(peerfd, &rset)) {
            if (client[i].revents & POLLIN) {
                char recvbuf[MAXLINE + 1] = { 0 };
                int ret = readline(peerfd, recvbuf, MAXLINE);
                if (ret == -1) {
                    ERR_EXIT("readline");
                }
                if (ret == 0) {
                    fputs("client close\n", stdout);
                    //FD_CLR(peerfd, &allset);
                    close(peerfd);
                    client[i].fd = -1;
                    continue;
                }
                fprintf(stdout, "receive: %s", recvbuf);
                //sleep(4);
                writen(peerfd, recvbuf, strlen(recvbuf));
                //write(peerfd, "test\n", strlen("test\n"));

                if (--nready <= 0) {
                    break;
                }

            }

        }

    }

//  close(peerfd);
    close(listenfd);

    return 0;
```

```
}
```

## epoll 编写服务器

```
int main(int argc, char **argv) {

    signal(SIGPIPE, SIG_IGN);
    int listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd < 0) {
        ERR_EXIT("socket");
    }

    int on = 1;
    if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0)
        ERR_EXIT("setsockopt");

    struct sockaddr_in servaddr;
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(8989);
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    socklen_t len = sizeof servaddr;
    int ret = bind(listenfd, (struct sockaddr*) &servaddr, len);
    if (ret < 0) {
        ERR_EXIT("bind");
    }

    ret = listen(listenfd, SOMAXCONN);
    if (ret < 0) {
        ERR_EXIT("listen");
    }


    int epollfd = epoll_create(EVENT_MAX);
    if (epollfd == -1) {
        ERR_EXIT("epoll_create");
    }
    struct epoll_event events[EVENT_MAX];
    int nready;
    struct epoll_event ev;
```

```
ev.data.fd = listenfd;
ev.events = EPOLLIN;
ret = epoll_ctl(epollfd, EPOLL_CTL_ADD, listenfd, &ev);
if (ret == -1) {
    ERR_EXIT("epoll_ctl");
}

while (1) {

    //nready = poll(client, maxi + 1, -1);
    nready = epoll_wait(epollfd, events, EVENT_MAX, -1);
    if (nready == -1) {
        if (errno == EINTR) {
            continue;
        } else {
            ERR_EXIT("epoll");
        }
    }
    if (nready == 0) {
        continue;
    }

    int i;
    for (i = 0; i < nready; ++i) {

        //if (client[0].revents & POLLIN) {
        if (events[i].data.fd == listenfd) {
            struct sockaddr_in peeraddr;
            bzero(&peeraddr, sizeof peeraddr);
            len = sizeof peeraddr;

            //accept
            int connfd = accept(listenfd, (struct sockaddr*) &peeraddr,
                    &len);
            if (connfd == -1) {
                ERR_EXIT("accept");
            }

            struct epoll_event ev;
            ev.data.fd = connfd;
            ev.events = EPOLLIN;
            int ret = epoll_ctl(epollfd, EPOLL_CTL_ADD, connfd, &ev);
            if(ret == -1){
                ERR_EXIT("epoll add");
```

```
                }


                fprintf(stdout, "IP = %s, port = %d\n",
                        inet_ntoa(peeraddr.sin_addr), ntohs(peeraddr.sin_port));

            } else {
                int peerfd = events[i].data.fd;
                if (peerfd == -1) {
                    continue;
                }
                //if (FD_ISSET(peerfd, &rset)) {
                if (events[i].events & POLLIN) {
                    char recvbuf[MAXLINE + 1] = { 0 };
                    int ret = readline(peerfd, recvbuf, MAXLINE);
                    if (ret == -1) {
                        ERR_EXIT("readline");
                    }
                    if (ret == 0) {
                        fputs("client close\n", stdout);
                        close(peerfd);
                        //client[i].fd = -1;
                        struct epoll_event ev;
                        ev.data.fd = peerfd;

                        ret = epoll_ctl(epollfd, EPOLL_CTL_DEL, peerfd, &ev);
                        if (ret == -1) {
                            ERR_EXIT("epoll_ctl");
                        }
                        continue;
                    }
                    fprintf(stdout, "receive: %s", recvbuf);
                    writen(peerfd, recvbuf, strlen(recvbuf));

                }
            }

        }

    }

//  close(peerfd);
    close(listenfd);
```

```
    return 0;
}
```