# C++

## 改变signal处理

struct sigaction sa;

struct kludge {

static void my\_sighandler(int sig) {

LogToFile::writeStringln("Ater-receive-sp-", "", "SIGPIPE happened"); //print something, LogToFile can be replaced by other print funcation.

}

};

sa.sa\_handler = kludge::my\_sighandler;

sigemptyset(&sa.sa\_mask);

sa.sa\_flags = 0;

if (sigaction(SIGPIPE, &sa, 0) == -1) {

LogToFile::writeStringln("Ater-receive-sp-", "", "sigaction failed");

}

## 3个不同的getline函数

std::[istream](http://www.cplusplus.com/reference/istream/istream/)::getline

istream& getline (char\* s, streamsize n );

istream& getline (char\* s, streamsize n, char delim );

std::getline ()

istream& getline (istream& is, string& str, char delim);

istream& getline (istream& is, string& str);

linux 提供：

ssize\_t getline(char \*\*lineptr, size\_t \*n, FILE \*stream);

#include <iostream> //cin, cout

#include <stdio.h> //printf， linux man getline

#include <cstddef> // size\_t, ssize\_t

#include <stdlib.h> // for free

int main() {

std::string name;

std::string movie;

std::cout<<"enter your name"<<std::endl;

**std::getline(std::cin, name);**

std::cout<<"enter your favourite movie"<<std::endl;

std::getline(std::cin, movie);

std::cout<<name<<"'s favourite movie is "<<movie<<std::endl;

std::cout << "enter something, man" << std::endl;

char s[100];

**std::cin.getline(s, sizeof(s));** //cin.getline has different syntax with std::getline

std::cout<<"s is "<<s<<std::endl;

char \*line = NULL;

size\_t n = 0;

ssize\_t read;

printf("please enter one line\n");

if ((read = **getline(&line, &n, stdin)**) != -1) {

printf("line is %s\n", line);

}

if (line) free(line);

}

~

## C++里面获得shell命令输出

#include <string>

#include <iostream>

#include <stdio.h>

std::string exec(char\* cmd) {

FILE\* pipe = popen(cmd, "r");

if (!pipe) return "ERROR";

char buffer[128];

std::string result = "";

while(!feof(pipe)) {

if(fgets(buffer, 128, pipe) != NULL)

result += buffer;

}

pclose(pipe);

return result;

}

## C++向文件中写入内容

FILE\* f\_hd = fopen("/home/review/X3App.txt", "w+");

const char \*s = "Yang Hongjun, stopX3App()....\n";

fwrite(s, strlen(s), 1, f\_hd);

fclose(f\_hd);

## 初始化一个struct

typedef struct MY\_TYPE {

boolean flag;

short int value;

double stuff;

} MY\_TYPE;

MY\_TYPE a = {0};

或者

My\_TYPE a; a = { true, 15, 0.123 };

或者

MY\_TYPE a = {true, 15, 0.123 };

或者

MY\_TYPE a = { .flag = true, .value = 15, .stuff = 0.123 };

注意1： 只要初始化struct里任一成员， 其他未被初始化的成员将被初始化为默认值。不论这个struct位于local还是global。

注意2：

如果struct位于local作用域， 其成员不会自动初始化，成员value为garbage 。

如果struct位于global作用域，或为static， 所有未被初始化成员被自动初始化为：

0：integers and floating point

‘\0’: char类型

NULL: for 指针

## 初始化数组的另一个方式

unsigned char msg[SEND\_BUFFER] = {0};

## {}的代替者

如if (condition) {

1;

2;

}

可以写为

If (condition) 1, 2;

## 恰当使用wrapper函数

## Strotol 字符串转换为不同格式的

printf("please input thread id:");

char s[100];

fgets(s, 100, stdin);

long threadId = strtol(s, NULL, 16) ;

## 打印函数的backtrace的方法

<http://stackoverflow.com/questions/3151779/how-its-better-to-invoke-gdb-from-program-to-print-its-stacktrace/4611112#4611112>

void show\_stackframe() {

#define SIZE 100

void \*buffer[SIZE];

char \*\*strings;

int npos = 0;

npos = backtrace(buffer, SIZE);

strings = backtrace\_symbols(buffer, npos);

if (strings == NULL) {

printf("Error happens in JsiTwConnection::show\_stackframe");

}

for (int j=0; j<npos; j++) {

printf("Hei\n");

printf("%s\n", strings[j]);

}

free(strings);

}

## 虚继承目的

istream 和 ostream 类对它们的基类进行虚继承。通过使基类成为虚基类，istream 和 ostream 指定，如果其他类（如 iostream 同时继承它们两个，则派生类中只出现它们的公共基类的一个副本。通过在派生列表中包含关键字 virtual 设置虚基类：

class istream : public virtual ios { ... };

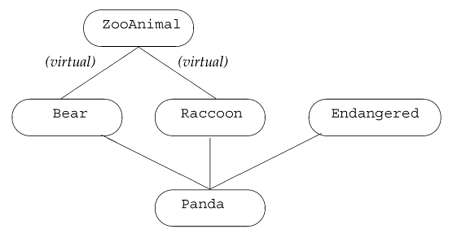
class ostream : virtual public ios { ... };

// iostream inherits only one copy of its ios base class

class iostream: public istream, public ostream { ... };

可见虚基类与否， 不是这个类自身赋予的，而是其他类赋予的。 虚基类（只允许一个实例）的特性，只有其他类多重继承2种以上类型的“虚基类的赋予者”（本例中istream/ostream）才会出现。

所以虚继承一个不直观的特征：必须在提出虚派生的任意实际需要之前进行虚派生（在例中，Bear 类和 Raccoon 类的虚派生）。只有在使用 Panda 的声明时，虚继承才是必要的，但如果 Bear 类和 Raccoon 类不是虚派生的，Panda 类的设计者就没有好运气了。



实际上，中间基类指定其继承为虚继承的要求很少引起任何问题。通常，使用虚继承的类层次是一次性由一个人或一个项目设计组设计的，独立开发的类很少需要其基类中的一个是虚基类，而且新基类的开发者不能改变已经存在的层次

## 虚基类的初始化

通常， 派生类只初始化自身的直接基类。 但这个原则不适用继承了虚基类的最终派生类， 因为常规原则，可能导致多次初始化虚基类。 所以在虚派生中， 由最终派生类的构造函数来初始化虚基类。这个最终是相对的， 所以基本上，有可能发挥虚基类特性的所有2层派生类（1层肯定初始化了）都需要在其构造函数中初始化虚基类了。如果没有在其构造函数中显式初始化虚基类，那么会调用虚基类的默认构造函数，如果虚基类没默认构造函数， 那么就会出现一个编译错误。

## 虚基类总是先于实基类被构造（不管继承先后）

## 多重继承时， 查找确定函数时， 不判断函数原型、access lable，只判断函数名即确定是否二义。

## 派生类的复制控制，析构控制

派生类的析构函数只负责清除派生类自身成员。 如果派生类定义了自己的复制构造函数和赋值操作符， 则要负责复制/赋值所有基类的子部分。

## C中的struct相较C++中类的优势

指向c中struct结构的指针， 可以对struct内的所有成员使用->, 如s->a, s->b。

但是C++中类的->操作符只能针对一个成员（一般选择指针数据成员）

## 头文件

通常头文件应该include implement文件需要用到的库， 但不要include implementation文件。

其他要使用implementation的代码直接include头文件即可。

但是在编译的时候，编译器如何去通过xx.h文件去查找同名的xx.cpp文件? 在Visual C++里面， 位于同一个project下，编译器能找到。

但是直接用G++调试，似乎不行的。方法有2个， 在编译使用头文件的程序时加上xx.cpp文件g++ basic.cpp ../../include/src/trim\_string.cpp -o basic.o。

或者为了方便， 直接在.h中include .cpp文件(实际代码不应该这样)。

## g++ and gcc

g++ is for C++.

gcc is for C, although with the –libstdc++, you can compile c++. But most people won’t take the way later.

。

## 程序的版式

### 每个类声明之后，每个函数定义结束之后都要加空行。

### 在一个函数体内， 逻辑上密切相关的语句之间不加空行，其他地方应该在空行分隔。

**while (condition)**

**{**

**statement1;**

**//** 空行

**if (condition)**

**{**

**statement2;**

**}**

**else**

**{**

**statement3;**

**}**

**//** 空行

**statement4;**

**}**

### 一行代码只做一件事情，如只定义一个变量，或只写一条语句。

### 像if、for、while等关键字之后应留一个空格再跟左括号’(‘, 以突出关键字

### 函数名之后不要留空格，紧跟左括号’(‘, 以与关键字区别

### ‘(‘ 向后紧跟， ‘)’ ‘，’ ‘;’ 向前紧跟， 不留空格。

### 对于表达式比较长的for语句和if语句，为了紧凑起见可以适当地去掉一些空格， 如for ( i=0; i<10; i++) 和if((a<=b) && (c<=d))

### 代码行最长长度宜控制在70至80个字符以内，代码行不要过长， 长表达式要在低优先级操作符出拆分为新行，操作符放在新行之首（以便突出操作符）。拆分出的新行要进行适当的缩进，使排版整齐，语句可读。

if ((very\_longer\_variable1 >= very\_longer\_variable12)

&& (very\_longer\_variable3 <= very\_longer\_variable14)

&& (very\_longer\_variable5 <= very\_longer\_variable16))

{

dosomething();

}

virtual CMatrix CMultiplyMatrix (CMatrix leftMatrix,

CMatrix rightMatrix);

for (very\_longer\_initialization;

very\_longer\_condition;

very\_longer\_update)

{

dosomething();

}

### 应将修饰符 \* 和 & 紧靠变量名， 但是对于函数的返回值， 我觉得还是紧靠数据类型比较好。

### 注释的位置应与被描述的代码相邻， 可以放在代码的上方或右方， 不可放在下方

## 命名规则

命名规则尽量与所采用的操作系统或开发工具的风格保持一致

例如windows应用程序的标识符通常采用”大小写“混排的方式， 如AddChild。

而Unix应用程序的标识符通常采用”小写加下划线“的方式， 如add\_child.别把这两类风格混在一起用。

### 变量和参数用小写字母开头的单词组合而成。常量全用大写的字母，用下划线分割单词， 静态变量加前缀s\_, 如果不得已需要全局变量，则使全局变量加前缀g\_, 类的数据成员加前缀m\_, 这样可以避免数据成员与成员函数的参数同名。

### 程序中不要出现仅靠大小写区分的相似的标识符

### 程序中不要出现标识符完全相同的局部变量和全局变量， 尽管两者的作用域不同不会发生语法错误，但会使人误解。

### 变量的名字应当使用“名词“或者”形容词+名词“

## 复合表达式

如a = b = c = 0这样的表达式称为复合表达式。违反一行只做一件事情允许复合表达式存在的理由是： 书写简洁， 可以提高编译效率。

不要把程序中的复合表达式与“真正的数学表达式”混淆。

例如：

if (a < b < c) // a < b < c 是数学表达式而不是程序表达式

并不表示

if ((a<b) && (b<c))

而是成了令人费解的

if ( (a<b)<c )

## 浮点变量与零值比较

不可将浮点变量用”==” 或”!=” 与任何数字比较。

千万要留意， 无论是float还是double类型的变量，都有精度限制。所以一定要避免将浮点变量用”==” 或“！=”与数字比较， 应设法转化成”>=” 或 “<=” 形式。

If (x == 0.0) //隐含错误的比较

转化为

If (( x>=-EPSINON) && (x<=EPSINON)), EPSINON是允许的误差（即精度）。

## If/else/return组合的编写风格

if (condition)

return x;

return y;

不好

应改写为

if (condition)

{

return x;

}

else

{

return y;

}

## 循环语句的效率

### 在多重循环当中，如果有可能， 应当将最长的循环放在最内存，最短的循环放在最外层， 以减少CPU跨切循环层的次数。

### 如果循环体内存在逻辑判断并且循环次数很大， 宜将逻辑判断移到循环体的外面。

示例4-4(c)的程序比示例4-4(d)多执行了N-1 次逻辑判断。并且

由于前者老要进行逻辑判断，打断了循环“流水线”作业，使得编译器不能对循环

进行优化处理，降低了效率。如果N 非常大，最好采用示例4-4(d)的写法，可以提

高效率。如果N 非常小，两者效率差别并不明显，采用示例4-4(c)的写法比较好，

因为程序更加简洁。

for (i=0; i<N; i++)

{

if (condition)

DoSomething();

else

DoOtherthing();

}

表4-4(c) 效率低但程序简洁

if (condition)

{

for (i=0; i<N; i++)

DoSomething();

}

else

{

for (i=0; i<N; i++)

DoOtherthing();

}

表4-4(d) 效率高但程序不简洁

### 优选半开半闭区间进行循环

for (int x=0; x<N; x++)

{

⋯

}

示例4-5(a) 循环变量属于半开半闭区间， 优选此种。

for (int x=0; x<=N-1; x++)

{

⋯

}

示例4-5(b) 循环变量属于闭区间

## 常量

#define MAX 100 /\* C 语言的宏常量 \*/

const int MAX = 100; // C++ 语言的const 常量

const float PI = 3.14159; // C++ 语言的const 常量

## 常量的存储规则

需要对外公开的常量放在头文件中，不需要对外公开的常量放在定义文件的头部。为了便于管理， 可以把不同模块的常量集中存放在一个公共的头文件中。

如果某一常量与其他常量密切相关，应在定义中包含这种关系，而不是给出孤立的值。

例如：

**const float RADIUS = 100;**

**const float DIAMETER = RADIUS \* 2;**

## 函数参数的规则

函数声明时，不要贪图省事只写参数的类型而省略参数名字：

例如：

void SetValue(int width, int height); // 良好的风格

void SetValue(int, int); // 不良的风格

## 函数的返回值规则

C语言中，凡不加类型说明的函数， 一律自动按整型处理。这样做不会有什么好处，却容易被误解为了void类型。C++有严格的类型安全检查，不允许上述情况发生，但由于C++程序可以调用C函数，为了避免混乱，规定任何C++/C函数都必须有类型，如果函数没有返回值，那么应声明为了void类型。

不要将正常值和错误类型混在一起返回。 正常值用输出参数获得，而错误标志用return语句返回。

## 函数内部实现的规则

### 在函数体的“入口处“， 对参数的有效性进行检查。

原因： 很多程序错误是由非法参数引起的， 我们应该充分理解并正确使用“断言“来防止此类错误。

### 在函数体的“出口处“，对return语句的正确性和效率进行检查

Return语句不能返回指向“栈内存“的指针或者引用， 因为该内存在函数体结束时被自动销毁。

如果函数的返回值是一个对象， 要考虑return语句的效率。

例如return String(s1 + s2); 这是临时对象的语法，表示“创建一个临时对象并返回它“。 不要以为它

String temp(s1 + s2);

Return temp;

是一样的。

前者创建一个临时对象并返回它。 编译器直接临时对象创建并初始化在外部存储单元中，省去了拷贝和析构的花费。

而后者先要创建一个临时对象，然后完成初始化，然后拷贝构造函数把temp拷贝到保存返回值的外部存储单元中；最后temp在函数结束时被销毁。

类似地，我们不要将

return int(x + y); // 创建一个临时变量并返回它

写成

int temp = x + y;

return temp;

由于内部数据类型如**int,float,double** 的变量不存在构造函数与析构函数，虽然该“临

时变量的语法”不会提高多少效率，但是程序更加简洁易读。

### 尽量避免函数带有“记忆”功能。相同的输入应当产生相同的输出。

带有“记忆”功能的函数，其行为可能是不可预测的，因为它的行为可能取决于某

高质量C++/C 编程指南，v 1.0

2001 Page 41 of 101

种“记忆状态”。这样的函数既不易理解又不利于测试和维护。在**C/C++**语言中，函数

的**static** 局部变量是函数的“记忆”存储器。建议尽量少用**static** 局部变量，除非必需。

### 不仅要检查输入参数的有效性，还要检查通过其它途径进入函数体内的变量的有效性，例如全局变量、文件句柄等。

## 为了避免函数调用时对内存的操作， 所以assert被设计为宏，而非函数。 当条件为假时，被触发

## 断言扑捉的是非法情况。 不要混淆非法情况和错误情况。

错误情况是必然出现的，并且一定要做出处理的。

在编写函数时，要进行反复的考查，并且自问：”我打算做哪些假定“， 一旦确定了假定， 就要使用断言对假定进行审查。

## . 优先级高于\*

## Cout char\*指针和int\* 指针的区别

int \*p = new int(1);

char \*str = new char[10];

strcpy(str, "funny");

cout<<\*p<<endl;

cout<<p<<endl;

cout<<str<<endl;

输出为

1

0x192a010

Funny

为什么cout<<str<<endl;可以输出内容呢， 因为对于指向字符串的指针，cout时，不是打印指针值， 而是打印该指针指向的内容直到遇到’\0’。

## 引用和指针的区别

引用定义时即需要赋值，不能有NULL的引用， 引用必须与合法的存储单元关联。（指针可以是NULL）. 一旦引用被初始化，就不能改变引用的关系（指针则可以随时改变所指的对象）。

引用(reference)是被引用物(referent)的别名/绰号。指针是被指对象的地址。当引用作为函数的参数时，传递的是referent的别名，称为引用传递。 当指针作为函数的参数时，传递的是地址，称为地址传递。

*void swap(A& v1, A& v2) {*

*A& temp = v1;*

*v1 = v2;*

*v2 = temp;*

*}*

*这个函数能如愿交换V1和V2的值吗？ 答案是不能， 函数执行完毕v1， v2的值都为v2.*

*class A {*

*public:*

*A(int v, double d): m\_v(v), m\_d(d) {}*

*A(): m\_v(0), m\_d(0.0) {}*

*int m\_v;*

*double get\_double() {*

*return m\_d;*

*}*

*private:*

*double m\_d;*

*};*

*int main() {*

*A A3(3, 3.0);*

*A A4(4, 4.0);*

*cout << "A3.m\_v is " << A3.m\_v << " at first" << ", A3.m\_d is " << A3.get\_double() << " at first" << endl;*

*A& r = A3;*

*r = A4;*

*cout << "A3.m\_v is " << A3.m\_v << " later" << ", A3.m\_d is " << A3.get\_double() << " later" << endl;*

*} 10:37*

*A3.m\_v is 3 at first, A3.m\_d is 3 at first*

*A3.m\_v is 4 later, A3.m\_d is 4 later*

*输出如上 1*

*A3的public和private的成员都被改变了*

*可以通过这种方式来破坏类的封装*

*但这不可能的*

*是不是： C和C++中一个变量的名字实际上就是一个指针*

*A A3(3, 3.0);*

*这句话在栈中申请了一个内存， 然后用3，3.0初始化。 A A4(4, 4.0);这个一样*

*main函数执行完毕后A3通过r被指向A4同样的地址了*

*A3原指向的内存里面的内容并没有被改写*

*可以通过下面代码验证*

*int main() {*

*A A3(3, 3.0);*

*A A4(4, 4.0);*

*A\* p = &A3;*

*cout << "Memory address of A3 is " << &A3 << " at first" << endl;*

*cout << "Memory address of A4 is " << &A4 << " at first" << endl;*

*cout << "A3.m\_v is " << A3.m\_v << " at first" << ", A3.m\_d is " << A3.get\_double() << " at first" << endl;*

*A& r = A3;*

*r = A4;*

*cout << "A3.m\_v is " << A3.m\_v << " later" << ", A3.m\_d is " << A3.get\_double() << " later" << endl;*

*cout << "Memory address of A3 is " << &A4 << " later" << endl;*

*cout << "p is " << p << endl; cout << "(\*p).m\_v is " << (\*p).m\_v << ", (\*p.m\_d) is " << (\*p).get\_double() << endl;*

*}*

*输出为：*

*Memory address of A3 is 0x7ffff2cce910 at first*

*Memory address of A4 is 0x7ffff2cce900 at first*

*A3.m\_v is 3 at first, A3.m\_d is 3 at first*

*A3.m\_v is 4 later, A3.m\_d is 4 later*

*Memory address of A3 is 0x7ffff2cce900 later*

*p is 0x7ffff2cce910*

*(\*p).m\_v is 4, (\*p.m\_d) is 4*

## 内存分配方式有三种

（**1**） 从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的

整个运行期间都存在。例如全局变量，**static** 变量。

（**2**） 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函

数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集

中，效率很高，但是分配的内存容量有限。

（**3**） 从堆上分配，亦称动态内存分配。程序在运行的时候用**malloc** 或**new** 申请任意

多少的内存，程序员自己负责在何时用**free** 或**delete** 释放内存。动态内存的生存

期由我们决定，使用非常灵活，但问题也最多。

## 用字符串初始化数组时， 字符串的最后一个隐含’\0’也被加入。

## 指针和数组的对比

**C++/C** 程序中，指针和数组在不少地方可以相互替换着用，让人产生一种错觉，以

为两者是等价的。

数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。**数组名对应着**

**（而不是指向）一块内存**，其地址与容量在生命期内保持不变，只有数组的内容可以改

变。如下

int main() {

//initial, 2015.7.22

int a[] = {1, 2, 3, 4};

int \*pa = a;

int (\*ptr) [4] = &a;

printf("a is %p, &a is %p, pa is %p, &pa is %p, &ptr is %p\n", a, &a, pa, &pa, &ptr);

char \*s = "funny";

printf("s address is %p, s is %s, &s is %p, &s[1] is %p, \*s is %c\n", s, s, &s, &s[1], \*s);

}

如上打印为：

*a is* *0x7fffdcce8220, &a is 0x7fffdcce8220, pa is 0x7fffdcce8220, &pa is 0x7fffdcce8218, &ptr is 0x7fffdcce8210*

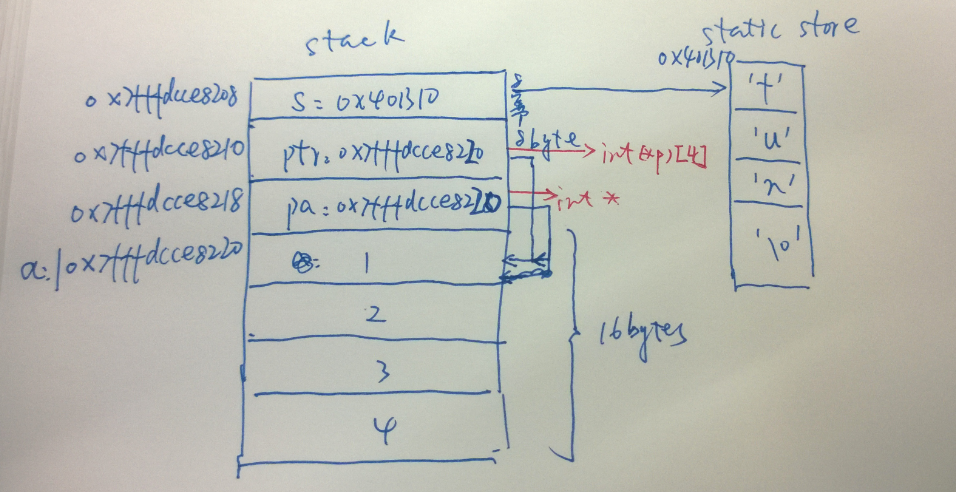
*s address is 0x40130f, s is fun, &s is 0x7fffdcce8208, &s[1] is 0x401310, \*s is f*

a is *0x7fffdcce8220*。 数组名将decay into a pointer who points to address of first element of array(只有两种例外，1. Sizeof(a), 2. &a), 所以在printf中a实际上指向数组a的第一个元素的指针。 %p printf打印指针会显示指针指向的地址， 故会显示栈上首地址。

&a is *0x7fffdcce8220*。 &a时：此处apply上面的两种例外之一，a不会decay 为一个指向int的指针，而为一个数组，而一个数组对应的就是一块内存， 而非指向一块内存， &取的地址就是这块内存的地址。所以&a不是指针的指针， 而为一个指向数组的指针， 类型为int (\*ap)[4]。%p printf打印指针会显示指针指向的地址， 故也会显示栈上首地址。

pa is *0x7fffdcce8220*. Pa是指针， 指向数组a的首元素。所以printf %p出来为数组a的首元素地址。

&pa地址为pa指针的地址为0x7fffdcce8218。 图如下：



**7.3.1** 修改内容

示例 **7-3-1** 中，字符数组**a** 的容量是**6** 个字符，其内容为**hello\0**。**a** 的内容可以改

变，如**a[0]= ‘X’**。指针**p** 指向常量字符串“**world**”（位于静态存储区，内容为**world\0**），

常量字符串的内容是不可以被修改的。从语法上看，编译器并不觉得语句**p[0]= ‘X’**有什

么不妥，但是该语句企图修改常量字符串的内容而导致运行错误。

char a[] = **“**hello**”**;

a[0] = **‘**X**’**;

cout << a << endl;

char \*p = **“**world**”**; // 注意p 指向常量字符串

p[0] = **‘**X**’**; // 编译器不能发现该错误

cout << p << endl;

**7.3.2** 内容复制与比较

不能对数组名进行直接复制与比较。示例7-3-2 中，若想把数组a 的内容复制给数

组b，不能用语句 b = a ，否则将产生编译错误。应该用标准库函数strcpy 进行复制。

同理，比较b 和a 的内容是否相同，不能用if(b==a) 来判断，应该用标准库函数strcmp

进行比较。

语句 p = a 并不能把a 的内容复制指针p，而是把a 的地址赋给了p。要想复制a

的内容，可以先用库函数malloc 为p 申请一块容量为strlen(a)+1 个字符的内存，再

用strcpy 进行字符串复制。同理，语句if(p==a) 比较的不是内容而是地址，应该用库

函数strcmp 来比较。

// 数组**…**

char a[] = "hello";

char b[10];

strcpy(b, a); // 不能用 b = a;

if(strcmp(b, a) == 0) // 不能用 if (b == a)

**…**

// 指针**…**

int len = strlen(a);

char \*p = (char \*)malloc(sizeof(char)\*(len+1));

strcpy(p, a); // 不要用 p = a;

if(strcmp(p, a) == 0) //不能用if(p == a)

…

**7.3.3** 计算内存容量

用运算符sizeof 可以计算出数组的容量（字节数）。示例7-3-3（a）中，sizeof(a)

的值是12（注意别忘了**’**\0**’**）。指针p 指向a，但是sizeof(p)的值却是4。这是因为

sizeof(p)得到的是一个指针变量的字节数，相当于sizeof(char\*)，而不是p 所指的内

存容量。C++/C 语言没有办法知道指针所指的内存容量，除非在申请内存时记住它。

注意当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。示例

7-3-3（b）中，不论数组a 的容量是多少，sizeof(a)始终等于sizeof(char \*)。

char a[] = "hello world";

char \*p = a;

cout<< sizeof(a) << endl; // 12 字节

cout<< sizeof(p) << endl; // 4 字节

示例 **7-3-3**（**a**） 计算数组和指针的内存容量

void Func(char a[100])

{

cout<< sizeof(a) << endl; // 4 字节而不是100 字节

}

示例**7-3-3**（**b**） 数组退化为指针

## 不能在类的声明中初始化const 常量

不能在类声明中初始化const 数据成员。以下用法是错误的，原因是如果此时允许对他的赋值， 那么在构造函数里面会再次对const 常量进行初始化。 而const常量的初始化有且仅有一次， 且定义时必须初始化。

class A

{⋯

const int SIZE = 100; // 错误，企图在类声明中初始化const 数据成员

int array[SIZE]; // 错误，未知的SIZE

};

class A

{⋯

A(int size); // 构造函数

const int SIZE ;

};

A::A(int size) : SIZE(size) // 构造函数的初始化表

{

⋯

}

A a(100); // 对象 a 的SIZE 值为100

A b(200); // 对象 b 的SIZE 值为200

怎样才能建立在整个类中都恒定的常量呢？别指望const 数据成员了，应该用类中

的枚举常量来实现。例如

class A

{⋯

enum { SIZE1 = 100, SIZE2 = 200}; // 枚举常量

int array1[SIZE1]; //在编译时SIZE1的值即已知，所以没问题的

int array2[SIZE2];

}

枚举常量不会占用对象的存储空间，它们在编译时被全部求值。枚举常量的缺点是：

它的隐含数据类型是整数，其最大值有限，且不能表示浮点数（如PI=3.14159）。

## 指针参数是如何传递内存的？

如果函数的参数是一个指针，不要指望用该指针去申请动态内存。示例**7-4-1** 中，

Test 函数的语句GetMemory(str, 200)并没有使str 获得期望的内存，str 依旧是NULL，

为什么？

void GetMemory(char \*p, int num)

{

p = (char \*)malloc(sizeof(char) \* num);

}

void Test(void)

{

char \*str = NULL;

GetMemory(str, 100); // str 仍然为 NULL

strcpy(str, "hello"); // 运行错误

}

示例**7-4-1** 试图用指针参数申请动态内存

高质量C++/C 编程指南，v 1.0

2001 Page 48 of 101

毛病出在函数GetMemory 中。编译器总是要为函数的每个参数制作临时副本，指针

参数p 的副本是 \_p，编译器使 \_p = p。如果函数体内的程序修改了\_p 的内容，就导致

参数p 的内容作相应的修改。这就是指针可以用作输出参数的原因。在本例中，\_p 申请

了新的内存，只是把\_p 所指的内存地址改变了，但是p 丝毫未变。所以函数GetMemory

并不能输出任何东西。事实上，每执行一次GetMemory 就会泄露一块内存，因为没有用

free 释放内存。

如果非得要用指针参数去申请内存，那么应该改用“指向指针的指针”，见示例**7-4-2**。

void GetMemory2(char \*\*p, int num)

{

\*p = (char \*)malloc(sizeof(char) \* num);

}

void Test2(void)

{

char \*str = NULL;

GetMemory2(&str, 100); // 注意参数是 &str，而不是str

strcpy(str, "hello");

cout<< str << endl;

free(str);

}

示例**7-4-2** 用指向指针的指针申请动态内存

由于“指向指针的指针”这个概念不容易理解，我们可以用函数返回值来传递动态

内存。这种方法更加简单，见示例**7-4-3**。

char \*GetMemory3(int num)

{

char \*p = (char \*)malloc(sizeof(char) \* num);

return p;

}

void Test3(void)

{

char \*str = NULL;

str = GetMemory3(100);

strcpy(str, "hello");

cout<< str << endl;

free(str);

}

示例**7-4-3** 用函数返回值来传递动态内存

高质量C++/C 编程指南，v 1.0

2001 Page 49 of 101

用函数返回值来传递动态内存这种方法虽然好用，但是常常有人把**return** 语句用错

了。这里强调不要用**return** 语句返回指向“栈内存”的指针，因为该内存在函数结束时

自动消亡，见示例**7-4-4**。

char \*GetString(void)

{

char p[] = "hello world";

return p; // 编译器将提出警告

}

void Test4(void)

{

char \*str = NULL;

str = GetString(); // str 的内容是垃圾

cout<< str << endl;

}

示例**7-4-4 return** 语句返回指向“栈内存”的指针

用调试器逐步跟踪Test4，发现执行str = GetString 语句后str 不再是NULL 指针，

但是str 的内容不是**“**hello world**”**而是垃圾。

如果把示例**7-4-4** 改写成示例**7-4-5**，会怎么样？

char \*GetString2(void)

{

char \*p = "hello world";

return p;

}

void Test5(void)

{

char \*str = NULL;

str = GetString2();

cout<< str << endl;

}

示例**7-4-5 return** 语句返回常量字符串

函数 Test5 运行虽然不会出错，但是函数GetString2 的设计概念却是错误的。因

为GetString2 内的“hello world”是常量字符串，位于静态存储区，它在程序生命期

内恒定不变。无论什么时候调用GetString2，它返回的始终是同一个“只读”的内存块。

## 如果输入参数以值传递方式传递对象， 宜食用引用传递， 这样可以省去临时对象的构造和析构

## Return语句的效率

return String(s1 + s2);

这是临时对象的语法，表示“创建一个临时对象并返回它”。不要以为它与“先创建

一个局部对象**temp** 并返回它的结果”是等价的，如

String temp(s1 + s2);

return temp;

实质不然，上述代码将发生三件事。首先，**temp** 对象被创建，同时完成初始化；然

后拷贝构造函数把**temp** 拷贝到保存返回值的外部存储单元中；最后，**temp** 在函数结束

时被销毁（调用析构函数）。然而“创建一个临时对象并返回它”的过程是不同的，编

译器直接把临时对象创建并初始化在外部存储单元中，省去了拷贝和析构的化费，提高

了效率。

## Return语句的简洁性

不要将

return int(x + y); // 创建一个临时变量并返回它

写成

int temp = x + y;

return temp;

由于内部数据类型如**int,float,double** 的变量不存在构造函数与析构函数，虽然该“临

时变量的语法”不会提高多少效率，但是程序更加简洁易读。

## Static局部变量

这个变量只能一次初始化， 但是可以多次加/减

static int a = 1; //这个语句在函数中只会执行一次

a++; //这个语句在函数中可以被多次执行。

## 函数返回值

C语言中， 凡不加类型说明的函数， 一律自动按整型处理。 这样做没有任何好处。却容易被误解为void类型。

C++ 语言有很严格的安全检查， 不允许上述情况发生。 由于C++程序可以调用C函数，为了避免混乱， 规定任何C++/C函数都必须有类型， 如果函数没有返回值，那么应声明为void类型。

## 断言

断言是为了检查非法情况的， 而不是检查错误情况的。 错误情况必然存在的且一定要作出处理的。

## Malloc/free 与new/delete

Malloc/free ： 他们是C/C++的库函数。 用于申请/释放用于存储内部数据的动态内存。

New/delete： 是运算符。 对于非内部数据类型的对象而言， 光用malloc/free无法满足动态对象的要求。 对象创建时要自动执行构造函数，对象消亡之前要自动执行析构函数。由于malloc/free是库函数而不是运算符，不在编译器控制权限之内， 不能够把执行构造函数和析构函数的任务强加于malloc/free. 内部数据类型的”对象“没有构造和析构的过程， 对他们而言malloc/free和new/delete是等价的。

既然 new/delete 的功能完全覆盖了malloc/free，为什么C++不把malloc/free 淘

汰出局呢？这是因为C++程序经常要调用C 函数，而C 程序只能用malloc/free 管理动

态内存。

Free(p) 如果由于p的类型以及它所指向的内存的容量事先都是知道的，所以语句free(p)能正确的释放内存，如果p是NULL指针，那么free对p无论操作多少次都不会出问题。如果p不是NULL指针，那么free对p连续操作两次就会导致程序运行错误。

## 可以使用delete [] p来删除单个内存， 故优先使用delete []

## C++程序如何调用C函数

假设某个**C** 函数的声明如下：

void foo(int x, int y);

该函数被**C** 编译器编译后在库中的名字为**\_foo**，而**C++**编译器则会产生像**\_foo\_int\_int**

之类的名字用来支持函数重载和类型安全连接。由于编译后的名字不同，**C++**程序不能

直接调用**C** 函数。**C++**提供了一个**C** 连接交换指定符号**extern**“**C**”来解决这个问题。

例如：

extern “C”

{

void foo(int x, int y);

⋯ // 其它函数

}

或者写成

extern “C”

{

#include “myheader.h”

⋯ // 其它C 头文件

}

在调用foo这个函数时，告诉**C++**编译译器，函数**foo** 是个**C** 连接，应该到库中找名字**\_foo** 而不是找**\_foo\_int\_int**。

**C++**编译器开发商已经对**C** 标准库的头文件作了**extern**“**C**”处理，所以

我们可以用＃**include** 直接引用这些头文件。

## 并不是2个函数的名字相同就能构成重载

全局函数和类的成员函数同名不算

重载，因为函数的作用域不同。例如：

void Print(⋯); // 全局函数

class A

{⋯

void Print(⋯); // 成员函数

}

不论两个 Print 函数的参数是否不同，如果类的某个成员函数要调用全局函数

Print，为了与成员函数Print 区别，全局函数被调用时应加‘::’标志。如

::Print(⋯); // 表示Print 是全局函数而非成员函数

## C++中只能靠参数，不能靠返回值来区分重载函数。

## 重载， 覆盖， 隐藏

成员函数被重载的特征:

* 相同的名字且相同的范围
* 参数不同
* Virtual名字可有可无
* 重载的目的： 同一个类中相同的函数名字，由于调用时参数类型/个数不同， 而是执行不同的代码。 如不同的构造函数去构造不同的类实例。

覆盖： 指派生类函数覆盖基类函数， 特征是：

* 同名但范围不同，派生类里的函数覆盖基类里的函数
* 参数相同
* 返回值可稍有不同。 这里不同指，一个是基类的指针/引用， 另一个是派生类的指针/引用。
* 基类函数必须有virtual关键字。
* 覆盖的目的：派生类需要修改/增加基类的行为。

隐藏： 值派生类隐藏了基类里的同名函数，基类的函数被派生类同名函数屏蔽。 用处不多，可想象到的目的： 如果一个派生类有多个基类， 想通过本派生类指针调用函数时， 以本类里定义的函数为准。那么就使用隐藏这个手段。如何隐藏：

* 只要派生类里有定义同名的函数，且基类里的函数和派生类函数不满足覆盖的条件。==派生类定义了同名， 但函数参数不同的函数。或者派生类定义了同名，且参数相同函数， 但是基类里函数没有用virtual声明。

## C++支持函数在类外的重载

## 参数是否有默认实参不构成函数重载

## 值型参数是否为const不构成函数重载，但是引用/指针参数是否const构成函数重载。

Record lookup(Phone);

Record lookup(const Phone); // redeclaration

上面的区别仅在于是否将形参定义为 const。这种差异并不影响传递至函数的对象；第二个函数声明被视为第一个的重复声明。其原因在于实参传递的方式。复制形参时并不考虑形参是否为 const——函数操纵的只是副本。函数的无法修改实参。结果，既可将 const 对象传递给 const 形参，也可传递给非 const 形参，这两种形参并无本质区别。

值得注意的是，形参与 const 形参的等价性仅适用于非引用形参。有 const 引用形参的函数与有非 const 引用形参的函数是不同的。类似地，如果函数带有指向 const 类型的指针形参，则与带有指向相同类型的非 const 对象的指针形参的函数不相同。

## 不能基于指针是否const来实现函数的重载

f(int \*);

f(int \*const); // redeclaration

## C++中局部的名字屏蔽全局的名字，这项规则还是用于局部的变量名会屏蔽全局函数名

不但会有局部的函数名屏蔽全局函数名，局部变量名屏蔽全局变量名。还有局部变量名屏蔽全局函数名，局部函数名屏蔽全局变量名。

/\* Program for illustration purposes only:

\* It is bad style for a function to define a local variable

\* with the same name as a global name it wants to use

\*/

string init(); // the name init has global scope

void fcn()

{

int init = 0; // init is local and hides global init

string s = init(); // error: global init is hidden

}

## C++中局部的同名函数对全局函数是屏蔽而不是重载。

void print(const string &);

void print(double); // overloads the print function

void fooBar(int ival)

{

void print(int); // new scope: hides previous instances of print

print("Value: "); // error: print(const string &) is hidden

print(ival); // ok: print(int) is visible

print(3.14); // ok: calls print(int); print(double) is hidden

}

## C++ 中也有缺省值

### 参数缺省值只能出现在函数的声明中，而不能出现在定义体中

例如：

void Foo(int x=0, int y=0); // 正确，缺省值出现在函数的声明中

void Foo(int x=0, int y=0) // 错误，缺省值出现在函数的定义体中

{

⋯

}

为什么会这样？我想是有两个原因：一是函数的实现（定义）本来就与参数是否有

缺省值无关，所以没有必要让缺省值出现在函数的定义体中。二是参数的缺省值可能会

改动，显然修改函数的声明比修改函数的定义要方便。

### 如果函数有多个参数，参数只能从后向前挨个儿缺省

否则将导致

函数调用语句怪模怪样。

正确的示例如下：

void Foo(int x, int y=0, int z=0);

错误的示例如下：

void Foo(int x=0, int y, int z=0);

## 运算符

C++中， 可以用关键字operator加上运算符来表示函数，叫做运算符重载。例如两个复数相加函数：

Complex Add(const Complex &a, Complex &b);

可以用运算符重载来表示：

Complex operator + (const Complex &a, const Complex &b);

运算符与普通函数在定义时差不多，但在调用时的不同之处是： 对于普通函数，参数出现在圆括号内；对于运算符，参数出现在其左、右侧。例如

Complex a, b, c;

C = Add(a, b);

C = a +b;

当然也可以像调用普通函数样调用重载操作符, 如： operator+(a, b).

如果运算符被重载为全局函数，那么只有一个参数的运算符叫做一元运算符， 有2个参数的叫做二元运算符。

如果运算符被重载为类的成员函数，那么一元运算符没有参数，二元运算符只有一个参数， 因为对象自己成为了左侧参数。

从语法上讲，运算符既可以定义为全局函数，也可以定义为成员函数。定义规则如下：

所有的一元运算符 -------🡪 建议重载为成员函数

= , (), [], -> -----🡪 只能重载为成员函数

+=, -=, /=, \*=, &=, |=, ~=, %=, >>=, <<= ---🡪建议重载为成员函数

所有其他运算符----🡪建议重载为全局函数。

对于成员运算符， 调用方式如调用普通的成员函数: 指定运行函数的对象， 使用’.’ 或者 ‘->’ 操作符获取希望调用的函数，同时传递所需数目和类型的参数。如

Object1 += object2 // 隐式调用+=操作符，和内置+=调用方式一样，直观。

Object1.operator+=(object2) //如同调用普通的类里的成员函数一样。

## 由于不能重载内置类型的运算符， 所以重载运算符参数中， 至少有一个是枚举类型或类类型

## 除了函数调用操作符 operator() 之外，重载操作符时使用默认实参是非法的？

## Inline必须与函数定义体放在一起才能使函数成为内联， 仅将inline放在函数声明前面不起任何作用。

当然也可以在函数声明时候， 加上inline， 当用户在阅读函数的声明时，提示用户这个函数是内联的。

## 定义在类声明之中的成员函数将自动成为内联函数

但这个不是很好的编程风格。最好如下

//头文件

Class A {

Public:

Void foo(int x, int y);

….

}

//定义文件

Inline void A::foo(int x, int y) {

…..

}

## 内联函数使用的场合

函数体内代码比较少， 没有循环。 原因： 内联是用“直接复制函数代码“ 取代 ”函数调用“， 如果函数代码较长或者存在循环， 那么程序总代码量增大， 导致消耗更多的内存。

类的构造函数和析构函数容易让人误解成使用内联更有效。要当心构造函数和析构函数可能会隐藏一些行为，如”偷偷地” 执行了基类或成员对象的构造函数和析构函数。所以不要随便地将构造函数和析构函数的定义体放在类声明中。

## 类里的函数

每个类有且只有一个析构函数和一个赋值函数和一个拷贝构造函数。 可以有多个构造函数（包括一个拷贝构造函数，其他的称为普通构造函数），对于任意一个类A， 如果不想编写上述函数, 编译器将自动为A产生四个缺省的函数，如：

A(void); //缺省无参数构造函数

A(const A &a); //缺省的拷贝构造函数

A& operator=(const A &a); //缺省的赋值函数

~A(); //缺省的析构函数

对于缺省的无参数构造函数， 如果用户自定义了任一个构造函数（包括无参数，有参数的）， 那么编译器将不会自动产生。一般的构造函数可以有多个的（即重载一般构造函数）。另外一般构造函数的参数不必要涵盖所有类的数据成员。如下：

class bar {

private:

int m\_a;

double m\_b;

public:

bar(int a):m\_a(a) {}

};

对于赋值函数，析构函数，拷贝构造函数， 肯定有一个存在， 它是要么是编译器自动产生的，要么是用户自定义的。

## 3种必须在初始化列表里初始化的数据类型

不管有没有显示初始化列表， 构造函数都尝试对数据成员进行初始化。 所以对那些定义即需要初始化的数据类型，必须在初始化列表里初始化。所以

类的const常量

引用类型

没有默认构造函数的类类型

只能在初始化列表中被初始化，不能在函数体内用赋值的方式来初始化

## 对于内部数据类型的数据成员而言，由于在初始化列表和{}里的初始化方式几乎没有区别，但后者的板式似乎更加清晰。所以优选后者。

## 类里面的数据成员的构造顺序不受它们在初始化列表中的次序的影响，只由成员对象在类中声明的次序决定。

因为类的声明是唯一的，而类的构造函数可以有多个，因此会有多个不同次序的初始化表。如果成员函数按照初始化表的次序进行构造，这将导致析构函数无法得到唯一的逆序。

## 为什么类里数据成员有指针变量时， 不能让编译器合成拷贝构造函数和赋值函数//这个看的， 如果类里的指针变量需要分配heap内存的话，确实

class String

{

public:

String(const char \*str = NULL); // 普通构造函数

String(const String &other); // 拷贝构造函数

~ String(void); // 析构函数

String & operate =(const String &other); // 赋值函数

private:

char \*m\_data; // 用于保存字符串

};

以类string的两个对象a, b 为例， 假设a.m\_dataa的内容为”hello”, b.m\_data的内容为”world”。 现将a赋给b，赋值函数的“位拷贝“意味着执行b.m\_data = a.m\_data。这将造成3个错误： 一是b.m\_data原有的内存没有被释放，造成内存泄露；二是b.m\_data和a.m\_data指向同一块内存，a和b任何一方变动都会影响另一方；三是在对象被析构时，m\_data被释放了两次。

这个看的m\_data是怎么初始化的：

-bash-4.1$

-bash-4.1$

-bash-4.1$

-bash-4.1$ cat pointerMemberOfClass\_in\_heap\_or\_stack.cpp

#include <iostream>

#include <string.h>

using namespace std;

class foo {

public:

foo(char\* pd): m\_data(pd) {}

void changeData(char\* str) { m\_data = str;}

void changeData(size\_t pos, char newc) { \*(m\_data+pos) = newc;}

void printData() { cout << m\_data << endl; }

protected:

char\* m\_data;

};

class bar {

public:

bar() {

m\_data = new char[100];

memset(m\_data, 0, 100);

}

void changeData(char\* str) { memcpy(m\_data, str, strlen(str));}

void printData() { cout << m\_data << endl; }

~bar() {

if (NULL != m\_data) { cout << "I am deleting\n"; delete [] m\_data; };

}

protected:

char\* m\_data;

};

int main(int argc, char\* argv[]) {

char p[] = {'g','a','m','e', '\0'};

char p2[] = {'f', 'u', 'n', '\0'};

foo f1(p);

foo f2 = f1;

f1.changeData(p2); //won't change f2.m\_data since f1.changeData only make f1.m\_data pointe to other place.

f1.printData();

f2.printData();

f2 = f1;

f1.changeData(0,'F'); //change f2.m\_data too since f1.m\_data and f2.m\_data point to same place.

f1.printData();

f2.printData();

bar b1, b2;

b1.changeData("game");

b2.changeData("fun");

b1.printData();

b2.printData();

b2 = b1; //memory leak here, b2.m\_data which was allocated at b2.changeData("fun") is leaking.

b1.changeData("GAME"); //b1.m\_data changed, cause b2.m\_data change since they are pointing to same place.

b1.printData();

b2.printData();

//b1.m\_data will delete twice during destructor"

}

输出如下：

-bash-4.1$ ./pointerMemberOfClass\_in\_heap\_or\_stack.o

fun

game

Fun

Fun

game

fun

GAME

GAME

I am deleting

I am deleting

\*\*\* glibc detected \*\*\* ./pointerMemberOfClass\_in\_heap\_or\_stack.o: double free or corruption (fasttop): 0x0000000001725010 \*\*\*

======= Backtrace: =========

/lib64/libc.so.6[0x3836675e66]

## 内置和复合类型成员的初始化

内置或复合类型的成员的初始值依赖于对象的作用域：在局部作用域中这些成员不被初始化，而在全局作用域（全局变量）中它们被初始化为 0。

cpp/class/build-in\_compound\_type\_initialize\_in\_synthetize\_constructor.cpp

#include <iostream>

using namespace std;

class people {

public:

void output() {

if (!name) {

cout<<"p.name is NULL"<<endl;

}

cout<<"age is "<<age<<", name is "<<name<<endl;

}

private:

int age;

char \*name;

};

people P;

int main() {

people p;

cout<<"output local p"<<endl;

p.output();

cout.clear(); //不能cout一个空指针

cout<<"output global P"<<endl;

P.output();

}

输出为

output local p

p.name is NULL //照理来说这个p不会被初始化为NULL指针，P此时应该随意指向任何地址。

age is -1813262432, name is output global P //age为初始化， 值是任意的。

p.name is NULL

age is 0, name is -bash-4.1$ //对于全局变量P， 其里面的age被初始化为0了

## 类初始化时调用构造函数或者拷贝构造函数，初始化后再次赋值调用赋值构造函数。

String a(“hello”);

String b(“wolrd”);

String c = a; //调用拷贝构造函数，风格较差。最好写成c(a)

c = b; //调用了赋值构造函数

## 基类的构造函数，析构函数，赋值函数不能被派生类继承。

原因简单： 不可能使用基类同样的方法去构造，析构，赋值派生类。 派生类比基类要多些数据成员。

## Virtual 函数

保留字:virtual 的目的是启用动态绑定。 成员函数默认为非虚函数，对非虚函数的调用在编译时确定。 所以基类通常应将派生类需要重定义的任意函数定义为虚函数。

除了构造函数之外， 任意非static成员函数都可以是虚函数。

保留字只在类内部的成员函数声明中出现，不能用在类定义体外部出现的函数定义上。

派生类中的虚函数的声明必须与基类中的定义方式完全匹配，但有一个例外：返回对基类型的引用（或指针）的虚函数。派生类中的虚函数可以返回基类函数所返回类型的派生类引用和指针。

一旦函数在基类中声明为虚函数，那么它就一直为虚函数，派生类无法改变该函数为虚函数这一事实。派生类重定义虚函数时，可以使用virtual保留字，但不是必须这样做。

## 基类的析构函数必须设置为虚函数

#include <iostream.h>

class Base

{

public:

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

};

class Derived : public Base

{

public:

virtual ~Derived() { cout<< "~Derived" << endl ; }

};

void main(void)

{

Base \* pB = new Derived; // upcast

delete pB;

**}**

输出结果为：

**~Derived**

**~Base**

如果析构函数不为虚，那么输出结果为

**~Base**

**原因是：析构函数不定义为虚， 动态绑定不能启用， delete指向派生类但是类型定义为基类的指针时， 不会执行派生类的析构函数，仅仅基类的析构函数被启用。**

**例： derivedClass\_destructor\_shouldbe\_virtual.cpp**

## 已定义的类才可以用作基类。如果已经声明了某类， 但没有定义该类， 那么不能使用该类作为基类

## 声明派生类是不必要声明派生列表

Class A: public class B\_A; //错误

Class A; //正确

## 函数调用触发动态绑定的条件

1. 只有指定为虚函数的成员函数才能进行动态绑定。
2. 必须通过基类的引用或指针进行函数调用。

## 动态类型和静态类型

基类指针或引用的关键点在于动态类型和静态类型的区别

静态类型： 在编译时可知的引用类型和指针类型。

动态类型： 在运行时才知晓的引用或指针类型。

Base \*p = Derived;

p->func();

如果这个func()在基类中被定义为了virtual的， 那么它的静态类型为基类类型的指针， 但是动态类型却为继承类类型指针。

## 多态性

引用和指针的静态类型和动态类型可以不同， 这是C++ 用以支持多态性的基石。

通过基类引用或指针调用基类中定义的函数时，我们并不知道执行函数的对象的确切类型，执行函数的对象可以是基类类型的， 也可能是派生类类型的。 如果调用非虚函数， 无论实际对象是什么类型，都调用指针静态类型所属的函数。 如果调用虚函数，则直到运行时才能确定调用哪个函数， 运行的虚函数是引用所绑定或指针所指向的对象所属类型定义的版本。

从编写代码的角度看我们无需担心。只要正确地设计和实现了类，不管实际对象是基类类型或派生类型，操作都将完成正确的工作。

另一方面，对象是非多态的——对象类型已知且不变。对象的动态类型总是与静态类型相同，这一点与引用或指针相反。运行的函数（虚函数或非虚函数）是由对象的类型定义的。

|  |  |
| --- | --- |
| |  | | --- | | 只有通过引用或指针调用，虚函数才在运行时确定。只有在这些情况下，直到运行时才知道对象的动态类型。 | |

## 虚函数和默认实参

向任何函数一样，虚函数也可以具有默认实参。 但注意，默认**实参的值在编译时即确定**，所以与动态类型无关。 通过基类的引用/指针调用虚函数使用基类里定义的默认值，通过派生类指针/引用调用，使用派生类的默认值。

故如果同一虚函数的基类版本和派生类版本使用不同的默认实参几乎一定会引起麻烦的。调用类型为基类指针/引用但指向/绑定对象为派生类的话， 必然造成默认实参值与期望的不一致。

## 只有成员函数中的代码才应该使用作用域操作符覆盖虚函数机制 unclear best practice

## 在编写派生类的赋值函数时，不要忘记对基类的数据成员重新赋值

class Base

{

public:

**…**

Base & operate =(const Base &other); // 类Base 的赋值函数

private:

int m\_i, m\_j, m\_k;

};

class Derived : public Base

{

public:

**…**

Derived & operate =(const Derived &other); // 类Derived 的赋值函数

private:

int m\_x, m\_y, m\_z;

};

Derived & Derived::operate =(const Derived &other)

{

//（1）检查自赋值

if(this == &other)

return \*this;

//（2）对基类的数据成员重新赋值

Base::operate =(other); // 因为不能直接操作私有数据成员

//（3）对派生类的数据成员赋值

m\_x = other.m\_x;

m\_y = other.m\_y;

m\_z = other.m\_z;

//（4）返回本对象的引用

return \*this;

}

## 类的继承

如果逻辑上B是A的一种，并且A的所有功能和属性对B而言都有意义，则允许B继承A的功能和属性。 B is a A

## 类的组合

如果逻辑上A是B的“一部分“， 则不允许B从A派生， 而是要用A和其他东西组合出B。B has a A.

## Const修饰函数参数

### 如果输入参数采用“指针传递”，那么加const 修饰可以防止意外地改动该指针，

### 起到保护作用。

例如 StringCopy 函数：

void StringCopy(char \*strDestination, const char \*strSource);

其中strSource 是输入参数，strDestination 是输出参数。给strSource 加上const

### 如果输入参数采用”值传递“， 由于函数将自动产生临时变量用于复制该参数，该输入参数本来就无需保护，所以不要加const 修饰。

### 如果输入参数是非内部数据类型的， 应该避免将”值传递“，采用const 引用传递。

因为值传递的过程中，需要产生临时变量，而临时变量的构造，赋值，析构都将消耗时间。所以因采用引用传递， 但是引用传递可能改变参数的值，所以加上const。

### 对于内部类型的输入参数，不要将“值传递“的方式改为”const 引用传递”

因为内部数据类型不存在构造，析构的过程，而且复制也非常快，值传递和引用传递的效率几乎相当。所以用“const 引用传递”替代“值传递”即达不到提高效率的目的，又降低了函数的可理解性。

## Const修饰函数的返回值

### 如果返回值是指针类型

如果加上const标签， 那么不能通过该指针来来改变指针指向的对象。同样，不能将该指针赋值给一个非const类型的指针

### 如果返回值是值型

如果是值型的话， 由于函数会把返回值复制到外部临时存储单元中，加上const修饰没有任何价值

### 如果返回值是引用

使用要谨慎， 搞清楚函数究竟是想返回一个对象的“拷贝”还是仅仅返回“别名”即可，还有“别名”的真身会不会在函数调用结束后被销毁掉。

函数采用引用传递的场合并不多，这种方式一般出现在类的赋值函数中，目的是为了实现链式表达。

例如

class A

{⋯

A & operate = (const A &other); // 赋值函数

};

A a, b, c; // a, b, c 为A 的对象

⋯

a = b = c; // 正常的链式赋值

(a = b) = c; // 不正常的链式赋值，但合法

如果将赋值函数的返回值加const 修饰，那么该返回值的内容不允许被改动。上例

中，语句 a = b = c 仍然正确，但是语句 (a = b) = c 则是非法的。

## Const成员函数

任何不会修改数据成员的函数都应该声明为const类型。如果在编写const成员函数时，不慎修改了数据成员，或者调用了其他非const成员函数，编译器将指出错误。

## 不能cout NULL指针， 否则cout的badbit会被置位。

## 使用string替代const char \*

如果const char \*类型作为类里面的数据成员，可以将其替换为C++ 提供的string类类型，相较于前者， 后者无须显示初始化这个成员，且比较安全。

## 类的访问标号

用户对类内成员的访问权限

|  |  |  |  |
| --- | --- | --- | --- |
| **访问标号** | **类内部用户** | **类外部用户** | **继承类** |
| Public | √ | √ | √ |
| private | √ | × | × |
| protected | √ | × | √ derived.member OK的，但是base.member 不OK |

## 类的继承标号

派生类继承了基类后， 派生类得到了基类里的成员。这些继承过来的成员的访问标号如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **基类访问标号** | **派生类继承public标号** | **派生类继承protected标号** | **派生类继承private标号** |
| Public | public | protected | private |
| private | inaccessible | inaccessible | inaccessible |
| protected | protected | protected | private |

总结：

对于基类中private成员， 派生类自己都不能访问， 故谈不上给其他用户的访问标号

如果是public继承， 基类中public&protected成员原来具有什么什么样的访问标号，现在一样

如果是protected继承， 基类中public&protected成员全部变为protected标号。

如果是private继承， 基类中的public&private成员全部变为private标号。

## protected成员的特殊属性

实例一允许访问实例二里的protected成员。

/cpp/class/protected\_access\_among\_instance.cpp

#include <iostream>

#include <string>

using namespace std;

class Test {

public:

Test(const char \*str): name(str) {}

void output\_names(const Test &T) {

cout<<"my name is "<<name<<endl;

cout<<"my neighbour's name is "<<T.name<<endl;

}

protected:

string name;

};

int main() {

Test T1("Li Lei");

Test T2("jim");

T1.output\_names(T2);

}

输出如下：

my name is Li Lei

my neighbour's name is jim

这个规则同样适用于从基类继承过来的protected成员：

#include <iostream>

using namespace std;

class Base {

public:

Base(int age, const char \*str): m\_age(age), name(str) {};

virtual void output() {

cout<<"name is "<<name<<endl;

}

int m\_age;

protected:

string name;

};

class Derived : protected Base {

public:

Derived(int age, const char \*str): Base(age, str) {}

virtual void output() {

cout<<"name is "<<name<<endl;

cout<<"age is "<<m\_age<<endl;

}

string get\_base\_protected\_member() {

cout<<"Base::name is "<<Base::name<<endl; //can access its own base class's member, accutually, \

Base::name == this.name

return Base::name;

}

void output\_names(const Derived &d, const Base &b) {

cout<<"my own name is "<<name<<endl;

cout<<"name of other derive class is "<<d.name<<endl; //very interesting, can access other derived \

class's member.

// cout<<"name of other Base class is "<<b.name<<endl; //can't access other base class's protected \

member surelly.

}

void output\_ages(const Derived &d, const Base &b) {

cout<<"my neighbour age is "<<d.m\_age<<endl;

cout<<"The age of father of another neighbour is "<<b.m\_age<<endl;

}

};

int main() {

Base b(0, "girl");

Derived d1(13, "lucy");

Derived d2(11, "lily");

d1.output();

d1.get\_base\_protected\_member();

d1.output\_names(d2, b);

d1.output\_ages(d2, b);

}

输出为：

name is lucy

age is 13

Base::name is lucy

my own name is lucy

name of other derive class is lily

my neighbour age is 11

The age of father of another neighbour is 0

## 默认实参于函数中

默认实参的给予可以放在声明中，亦可以放在定义中。但是不能2处都放。

但最好在声明处给予默认实参， 原因是：别处要使用带默认实参的函数时，只需要include带这个函数声明的头文件即可，否则，还要include定义函数的源文件。

## Ios::tie

有2中用法

Ostream\* tie() : 返回调用这个函数的对象所tie的ostream。

Ostream\* tie( ostream\* tiestr); 将调用这个函数的对象tie到tiestr上去， 并且返回调用这个函数的对象以前所tie的ostream

默认情况， cin, cerr, clog 会tie到cout上的

+~/cpp/IO/tie.cpp

#include <iostream>

#include <fstream>

using namespace std;

int main() {

ostream \*p\_prevstr;

ofstream filestr;

filestr.open("test.txt");

\*cin.tie()<<"this is inserted into cout\n";

p\_prevstr = cin.tie(&filestr);

\*cin.tie()<<"this is inserted into file\n";

cin.tie(p\_prevstr);

filestr.close();

}

## Iostream当中的光标移动

注意，使用istream/ifstring/istringstream >> str，写入内容到str的时候， 光标是在不断移动的直到eof， 现在似乎没有方法将光标回到最初。方法只有重新创建上述3个对象。

另外一点如果上述的对象有错误， 要使用clear清除错误，才能再次使用

## Ifstream/ofstream

使用open(c\_str)打开返回的是none。

Ifstream(c\_str)，这个是构造函数，隐式返回的是对象自身。

判断对象有没有打开成功， 使用if(object).

## Getline

Std::getline 可以给string赋带空格的值，cout不行。

Iostream::getline语法如下：

istream& getline (char\* s, streamsize n );

istream& getline (char\* s, streamsize n, char delim );

第一个参数不是string类型， 且必须指定最大长度， optionally指定分隔符， 如果不指定为换行符。

#include <iostream>

int main() {

std::string name;

std::string movie;

std::cout<<"enter your name"<<std::endl;

std::getline(std::cin, name);

std::cout<<"enter your favourite movie"<<std::endl;

std::getline(std::cin, movie);

std::cout<<name<<"'s favourite movie is "<<movie<<std::endl;

// std::cin.getline(name)

char s[100];

std::cin.getline(s, sizeof(s)); //cin.getline has different syntax with std::getline

std::cout<<"s is "<<s<<std::endl;

}

## Sstream

Sstream用武之地似乎有2个地方：

* 1. 可以用一行值来初始化它， 然后使用>>来一个个的读取行中的单个字符。

利用这个， 可以每行每行的读取文件内容，然后一个字一个字的处理

+~/cpp/IO/sstring.cpp

#include <iostream>

#include <sstream>

#include <string>

using namespace std;

int main() {

string dump;

istringstream iss("hello world\n"); //istringstream can use line to initialize. \

But cin just input one single word

ostringstream oss;

while (iss >> dump) { //istringstream can write word one by one.

cout<<dump;

cout<<"#";

oss<<dump;

cout<<oss.str();

}

}

* 1. 写入ostringstream的类型数据，如果用ostringstream.str()来初始化istringstream， 然后使用istringstream写出的时候， 还能保持数据类型。 了解

## Vector容器的size可以运行是指定

#include <iostream>

#include <vector>

using namespace std;

int main() {

const int size = 10;

int size2;

cout << "please input a integer:";

cin >> size2;

vector<int> i\_vec(size+size2, 0);

for(vector<int>::iterator iter=i\_vec.begin(); iter<i\_vec.end(); iter++) {

cout<<\*iter<<endl;

}

}

## 只接受容器大小作为形参的构造函数只适用于顺序容器，而关联容器不支持这种初始化。

## 容器的元素类型需满足2个特点： 元素类型必须支持赋值操作。 元素类型的对象必须可以复制。

## 元素为容器的容器

Vector < vector<string> > 对的

Vector< vector<string>> 错的，必须使用空格隔开相邻的>符号，否则 >>会被认为右移操作符。

## Vector<xx>::size\_type 无符号整型， 足以存储此容器的最大可能容器长度。

## 容器迭代器相等条件

如果它们指向同一个容器内的同一个元素，或者指向同一个容器指向末端的下一个位置。

## 迭代器支持的操作

所有迭代器支持的操作： 解引用，箭头、自增/自减、等于/不等于。

因为vector和deque为其元素提供快速、随机的访问， 所以支持的额外算术和比较操作： +n/-n、iter1 – iter2、 >, >=, <, <=。

如果iter1指向元素位置是在iter2指向元素的前面，那么iter1就大于iter2。

## 对容器里的元素进行move/add/remove操作一定会导致指向被move/remove的迭代器失效， 甚至会导致所有指向该容器的迭代器失效。

## 在容器中添加元素时， 系统是将元素值复制到容器中。 类似的，使用一个容器初始化另一个容器，新容器里存储的是原始元素的副本。

## 使用insert向容器某个位置（结尾）插入一个元素的代价很高，因为要移动被插入位置之后所有的元素。当然， 向list里任何位置插入元素都很高效。

## Resize操作可能是vector/deque的所有迭代器失效（原因是存储容器的地址由于长度变化， 需要重分配，导致变化）。 对于所有容器，如果resize压缩了容器大小，则指向已删除的元素的迭代器失效。

## 引用对值型值赋值时，依然为值传递。 引用对引用赋值，才是引用传递。

## Deque和vector还有object.at[n]的方法。

返回下标为n的元素的引用----没什么大用，作用同object[n], 只不过前者如果下标n不存在的话， 会抛出out\_of\_range 的异常。

## Contrainer.erase()

Containter.erase(p) 返回的是p之后的迭代器。如果p是最后一个元素， 那么返回的迭代器指向超出容器末端的下一个位置。 p本身指向超出末端的下一个位置的话，则该函数未定义。

Container.erase(b, e) 返回指向被删除段后面的元素的迭代器， 但是 e允许指向超出末端的下一个位置

## 解引用或者输出空引用将会导致segmentation fault in run-time

//de-reference empty pointer will cause segement fault in run-time.

vector<char> cvec;

int \*p = NULL;

cout << \*p << endl; //segementation fault

cout << cvec.front() << endl; //segementation fault

cout << \*cvec.begin() << endl; //segementation fault

}

## Push\_front/ pop\_front、push\_back/pop\_back 函数返回void

## Vector1.swap(vector2)

这个函数实际是将vector1指向原vector2的地址， 反之亦然。 所以迭代器在swap后依然有效， 指向同样的地址。 但是vector1和vector2指向不同的地址了。 由于交换地址， 所以vector1和vector2的类型必须相同。

## Vector.assign(b, e)和vector.assign(n, t) assign也有2个版本， 哈哈

## 适配器基础概念

概念： 标准库提供的一个类， 本质上， 适配器是使一事物的行为类似于另一事物的行为的一种机制。

标准库里面有三种适配器: 容器适配器， 迭代器适配器， 函数适配器。 对于容器适配器来说，它让一种已经存在的容器类型行为类似于另一种不同的抽象类型， 如stack， queue， priority\_queue(成员在堆里面按照优先级排序， 排序时使用的默认比较函数为<, 最大的位于最前面)., 对于这三者，所用的适配器，分别为stack， queue， priority\_queue.

容器适配器具有的具有的成员：

|  |  |
| --- | --- |
| size\_type | Type large enough to hold size of largest object of this type.  一种类型，足以存储此适配器类型最大对象的长度 |
| value\_type | Element type.  元素类型 |
| container\_type | Type of the underlying container on which the adaptor is implemented.  基础容器的类型，适配器在此容器类型上实现 |
| A a; | Create a new empty adaptor named a.  默认构造函数， 创建一个新空适配器，命名为 a |
| A a(c); | Create a new adaptor named a with a copy of the container c.  带容器参数的构造函数。 创建一个名为 a 的新适配器，初始化为容器 c 的副本 |
| Relational Operators  关系操作符 | Each adaptor supports all the relational operators: ==, !=, <, <=, >, >=.  所有适配器都支持全部关系操作符：==、 !=、 <、 <=、 >、 >= |

使用适配器时， 需包含的相关头文件是：

#include <stack>

#include <queue>

## 适配器的定义， 覆盖基础容器类型

默认的stack和queue都是基于deque容器实现的， 而priority\_queue则在vector容器上实现。

故stack<int> stk(deq); 这个适配器只能用deque类别来初始化、priority\_queue<int> p\_que(vec)只能用vector来初始化。

但是在创建适配器时， 可以通过规定适配器的第二个类型参数来重写默认基础容器类型。

Stack < string , vector<string> > str\_stk;s

对于给定的适配器， 其关联的容器必须满足一定的约束条件。

Stack适配器所关联的基础容器可以是任意一种顺序容器类型。

Queue由于需要提供push\_front操作， 故其关联的基础容器不能为vector， list, deque行。 如果将vector作为queue的关联容器， 编译不会发生错误的， 但是使用pop()的时候会抛出错误： â€˜class std::vector<int, std::allocator<int> >â€™ has no member named â€˜pop\_frontâ€™

Priority\_queue 由于需要提供随机访问功能， 因此可以建立在vector和deque容器上，但不能建立在list上。

## Stack, queue, priority\_queue的概念及提供的操作

Stack不用解释了，他就是栈后进先出的行为。

s.empty(), s.size(), s.pop(), s.push(item)

s.top() 返回栈顶的元素， 但不删除它

Queue 是先进先出的队列行为

q.empty(), q.size(), q.pop(), q.push(item)

q.front(), q.back() 返回队列首尾元素，但不删除它们。

Priority\_queue是一个heap，heap的概念是经过排序了树了， 最高优先级的在最前面。它不要求push\_front的操作。

p.empty(), p.size(), p.pop(), p.push(item)

p.top() 返回堆顶的元素， 但不删除它。

## Vector和pair 空初始化采用值初始化(value-intialized)

Vector<int>, pair<string, int>

这样创建空的vector或pair。

对于类类型： 有默认构造函数，使用默认构造函数。 对于没有定义任何构造函数的类类型， 标准库仍然生成一个带初始值的对象， 这个对象里面的每个成员进行了值初始化。

，不同于普通内置变量的初始化。

对于内置类型： 用0值进行初始化（这点不同于普通变量的初始化，其方式要根据作用域来看的）。

## 关联容器相较顺序容器的操作

它不是顺序容器， 故无顺序， 关联容器没有什么头尾， 故什么front、back、push\_front、push\_back、 pop\_front、 pop\_back 也没有了。元素在容器中实际的顺序是什么完全没用， 例如遍历元素时， 是根据键的顺序来遍历的， 与元素在容器中的存放位置完全无关。

但是似乎有begin(), end(), rbegin(), rend() 操作，因为需要一个迭代器的起点和终点来遍历整个容器

又由于关联容器，如果键值已定， 不可能在改写键值了。 所以不能创建空的键值元素了，因为以后没用， 你没法改写。 所以初始化时， 不可能规定关联容器的大小，不初始化其元素， 即map<string, int> m(10) 这样的是不合法的。 再次推导值resize()也是没用的了。

Erase操作对于顺序容器，返回的是被erase元素后面的迭代器。由于关联容器没有顺序一说， 所以erase对于关联容器，返回void。但是新增了erase版本， m.erase(k) 将会删除map中所有key为k的元素。

Assign操作对于关联容器失效。 原因未知， 因为可以使用m(b, e)来初始化一个map，但是m.assign(b, e)却不行。 不去探知了。

Insert插入元素到关联容器中依然存在，但是语法改变了，下面详述

## map的初始化， 修改

* 初始化

Map<type1, type2> m 创建一个空的关联容器

Map<type1, type2> m(m2) 以m2来初始化m

Map<type1, type2> m(b, e). 以b,e 范围的迭代器指向的元素来初始化map， 当然b,e 指向对象类型要和pair<const type1, type2> 吻合。

* 修改：
* 直接下标修改, 如map<string , int> m这样的关联容器。 M[“test”] = 1这样来添加/修改， 如果”test”这个键值以前不存在， 那么添加这个键值对。如果存在的话， 修改test这个键对应的值。
* 使用insert来修改。由于关联容器没有顺序之分， 所以无需规定插入的位置了， 所以相较于顺序容器的插入， 无需第一个迭代器参数来指明插入位置了。
  + M.insert(e): e是一个键值对，如果键e.first不在m中， 则插入值为e.second的新元素。如果该键在m中已存在， 那么m保持不变。 该函数返回一个pair类型对象， 这个pair类型key为指向键为e.first的元素的map迭代器， value为bool类型，表示是否插入了该元素。
  + *M.insert(b, e): beg 和 end 是标记元素范围的迭代器，其中的元素必须为 m.value\_type 类型的键－值对。对于该范围内的所有元素，如果它的键在 m 中不存在，则将该键及其关联的值插入到 m。返回 void 类型*
  + M.insert(iter, e): 如果e.first不在m的键中， 插入之。返回refer to m中具有给定键的元素的迭代器， 不论是否插入成功。 书上对iter解释如下：inserts the new element using the iterator iter as a hint for where to begin the search for where the new element should be stored. 不理解是什么意思， 也不从程序中理解了。
* 删除
* M.erase(k) 删除k的键值对， 返回删除的键值对的个数。 在map当中，当然为1， 对于multi-map， 可能为多个。
* M.erase(p) 删除迭代器p指向的元素， 返回void。
* M.erase(b, e) 删除迭代器b,e 之间的元素，返回void

## Map容器的额外函数

m.cout(k) 查找m中key: k 出现的次数。

m.find(k) 查找m中key:k 的位置，如果k不存在，那么返回m.end();

## 快速创建pair的方法

Make\_pair(v, v) 还有种

Typedef map<string, int>::value\_type valType;

valType(“pair”, 10); 直接类型然后初始化即可。 直接调用pair<string, int> 的构造函数来创建对象的， of course, 这个对象接受一个string， 一个int类型的参数。

## 遍历map容器的元素是根据键值的比较来分先后的， 故map容器的元素的键值， 必须规定有< 操作符，且能正确的工作。

## Multimap 和multiset中，如果某个键对应多个实例，则这些实例在容器中将相邻存放。

## 返回迭代器的关联容器的操作

m.lower\_bound(k) : 返回一个迭代器，指向键不小于k的第一个元素

m.upper\_bound(k) : 返回一个迭代器， 指向键大于k的第一个元素

m.equal\_range(k): 返回一个pair对象，该对象的第一个成员等价于m.lower\_bound(k), 第二个成员等价于m.upper\_bound(k).

## 泛型算法中的find\_frist\_of接受2个迭代器对， 这2个iterator pair类型不需要匹配， 唯一要求是这2个iterator pair指向的元素可相互比较。

如find\_first\_of(roster1.begin(), roster1.end(), roster2.begin(), roster2.end() )

如果roster1 为vector<string>, roster2 为 list<char \*>, 也是没有问题的， 因为string标准库里面为string对象和char\*对象定义了相等(==)运算符。

## 插入器 和插入迭代器之间的关系

插入器它是一个函数， 这个函数接收一个container， 返回值为插入迭代器, 包括back\_insert\_iterator, front\_insert\_iterator, insert\_iterator.

上面这3种迭代器是C++标准库提供的另一种迭代器， 它们共同特点是：1. 对迭代器进行赋值操作， 将会向特定位置插入被赋值的元素。 2. 它们没有默认构造函数， 创建它们需要调用插入器来。

back\_insert\_iterator 可由back\_inserter(Container&)来创建， 对这个迭代器赋值将会调用Container的push\_back(v)函数向容器尾部添加一个元素。

原型为： template <class Container>

back\_insert\_iterator<Container> back\_inserter (Container& x);

Front\_insert\_iterator可由front\_inserter(Container&)来创建， 对这个迭代器赋值将会调用Container的push\_front(v)函数向容器头部添加一个元素。

原型： template <class Container>

front\_insert\_iterator<Container> front\_inserter (Container& x);

Insert\_iterator可由inserter(Container&, container::iterator)来创建， 对这个迭代器赋值将会调用Container的insert(p, v)函数向p迭代器之前添加一个元素。 注意可以一次性向insert\_iterator赋多个值， 这多个值插入位置是依然是p， 只不过每次插入后p++. 结果例如向元素5前面插入插入：1，2，3，4， 结果为1，2，3，4，5. 而非4，3， 2，1， 5.

函数原型为： template <class Container, class Iterator>

insert\_iterator<Container> inserter (Container& x, Iterator it);

## Ostream\_iterator没有超出末端的迭代器， 我想原因是“向流里面写入，是永不会停止的， 如果你愿意“， 所以也不能定义一个无初始化的ostream\_iterator. 而istream\_iterator是用来从流中读取目标类型的元素， 读流/文件 总有尽头的， 所以可以定义一个无初始化的istream\_iterator表示流到达结尾。

## 使用同一个流初始化Istream\_iterator2遍， 将会导致使用流的>> 2次

template <class T, class charT=char, class traits=char\_traits<charT>, class Distance=ptrdiff\_t>

class istream\_iterator :

public iterator<input\_iterator\_tag, T, Distance, const T\*, const T&>

{

basic\_istream<charT,traits>\* in\_stream;

T value;

public:

typedef charT char\_type;

typedef traits traits\_type;

typedef basic\_istream<charT,traits> istream\_type;

istream\_iterator() : in\_stream(0) {}

istream\_iterator(istream\_type& s) : in\_stream(&s) { ++\*this; }

istream\_iterator(const istream\_iterator<T,charT,traits,Distance>& x)

: in\_stream(x.in\_stream), value(x.value) {}

~istream\_iterator() {}

const T& operator\*() const { return value; }

const T\* operator->() const { return &value; }

istream\_iterator<T,charT,traits,Distance>& operator++() {

if (in\_stream && !(\*in\_stream >> value)) in\_stream=0;

return \*this;

}

istream\_iterator<T,charT,traits,Distance> operator++(int) {

istream\_iterator<T,charT,traits,Distance> tmp = \*this;

++\*this;

return tmp;

}

};

故

ifstream ifile("story.txt");

istream\_iterator<string> in(ifile);

istream\_iterator<string> in2(ifile);

cout << \*in << endl; //story.txt的第一个元素

cout << \*in2 << endl; //story.txt的第二个元素。

## Istream\_iterator支持的操作

解引用 \*： 返回迭代器关联的流>>出来的元素的引用拷贝。不能改变“解引用”， 因为const T& operator\*() const { return value; }， 或者另一个理由： 不能像输入迭代器里面写入内容。

自增操作 ++it, it++. whenever operator++ is used on the iterator, it extracts an element from the stream (using operator>>).

istream\_iterator<T,charT,traits,Distance>& operator++() {

if (in\_stream && !(\*in\_stream >> value)) in\_stream=0;

return \*this;

}

Operator->: 返回迭代器关联流>>出来的元素的指针。 同样不能通过这个指针改变其值。

const T\* operator->() const { return &value; }

比较运算符 !=, ==. 如果2个迭代器指向同一个流， 那么它们相等。任何2个eof的istream\_iterator相等。

## 避免使用同一个流创建多个istream\_iterator。避免对istream流使用>>的同时, 又对这个istream关联的istream\_iterator使用自增操作。 因为这些操作都会调用istream当中的>>, 这个操作符改变流的当前位置。

参见：+~/cpp/stream\_iterator/increase\_iterator.cpp

#include <iostream>

#include <iterator>

#include <fstream>

using namespace std;

int main() {

//story content:

//there was a temple long long ago

//there were a old monk and a little monk in this temple

ifstream ifile("story.txt");

if (!ifile) {

cout << "Open file failed" << endl;

return -1;

}

ofstream ofile("story\_copy.txt");

if (!ofile) {

cout << "Open file to write failed " << endl;

return -1;

}

istream\_iterator<string> in(ifile);

istream\_iterator<string> in2(ifile);

istream\_iterator<string> eof;

istream\_iterator<string> eof2;

ostream\_iterator<string> out(ofile);

string dump;

//use istream\_iterator to read stuff from istream

//normal way to read stuff from istream.

cout << \*in << endl; //there

cout << \*in2 << endl; //was

cout << \*++in << endl; //a

ifile >> dump; //temple

cout << dump << endl; //

cout << \*in << endl; //a

cout << \*++in << endl; //long

// \*in == "test"; //wrong. restrict: can't write into object pointed by istream\_iterator

//non-member function: ==, !=;

cout << "\*in2:" << \*in2 << endl; //was

cout << " \*in:" << \*in << endl; // long

cout << (in == in2) << endl; //1

cout << (eof == eof2) << endl; //1

## 任何定义了”<<”操作符的数据类型， 都可创建自己的istream\_iterator<type> it(istream)

istream\_iterator<Sales\_item> item\_iter(cin), eof;

## Ostream\_iterator

They are constructed from a [basic\_ostream](http://www.cplusplus.com/basic_ostream) object, to which they become associated, so that whenever an assignment operator (=) is used on the [ostream\_iterator](http://www.cplusplus.com/ostream_iterator) (dereferenced or not) it inserts a new element into the stream.

只要使用了赋值操作符， 那么就会向流里面插入一个元素。

存在解引用和自增操作， 但是这2个操作都是返回\*this， 没啥意义。

template <class T, class charT=char, class traits=char\_traits<charT> >

class ostream\_iterator :

public iterator<output\_iterator\_tag, void, void, void, void>

{

basic\_ostream<charT,traits>\* out\_stream;

const charT\* delim;

public:

typedef charT char\_type;

typedef traits traits\_type;

typedef basic\_ostream<charT,traits> ostream\_type;

ostream\_iterator(ostream\_type& s) : out\_stream(&s), delim(0) {}

ostream\_iterator(ostream\_type& s, const charT\* delimiter)

: out\_stream(&s), delim(delimiter) { }

ostream\_iterator(const ostream\_iterator<T,charT,traits>& x)

: out\_stream(x.out\_stream), delim(x.delim) {}

~ostream\_iterator() {}

ostream\_iterator<T,charT,traits>& operator= (const T& value) {

\*out\_stream << value;

if (delim!=0) \*out\_stream << delim;

return \*this;

}

ostream\_iterator<T,charT,traits>& operator\*() { return \*this; }

ostream\_iterator<T,charT,traits>& operator++() { return \*this; }

ostream\_iterator<T,charT,traits>& operator++(int) { return \*this; }

}

因为使用=操作符后， 就像流里面写入了内容，所以， 一旦对ostream\_iterator对象赋了值， 写入就提交了。 赋值后，没办法改变这个值了。

## 传递一对反向迭代器给sort， 会以降序排列元素。 不清楚为什么会这样

## 反向迭代器需要定义自减操作符， 由于不能反向遍历流， 所以流迭代器也没有自减操作， 故也不能创建流的反向迭代器。

## Find和find\_if不重载的原因

Find(beg, end, val)

Find\_if(beg, end, pred); pred是一个谓语函数。

这2个函数并没有重载， 因为如果重载的话， 函数形参个数相同， 可能造成函数调用的二义性。

## 一般的如果标准库提供了某个谓语(do)函数算法， 那么也提供令人do\_copy算法。 如unique/unique\_copy, reverse/reversr\_copy.

## List容器的迭代器由于不是随机访问类型的， 所以通用sort算法不能使用， 而merge， remove， reverse, unique算法虽然可用， 但是会付出性能上代价（和通用算法实现可能有关）。 此时标准库特意定义了针对list的算法。

Lst.merge(lst2)： 将lst2合并到lst, 合并时使用<排序, 小者在前。 返回空。 合并后lst2里的元素将被删除。

Lst.merge(lst2, comp)： 使用comp进行排序。 为真时的左者在前。

See +~/cpp/algorithm/list\_specific\_algorithm.cpp

## 句柄类

面向对象编程的动态靠“指针“来实现的， 而不是靠对象， “指针.method()”在运行期间会更具指针实际指向的对象而选择基类或者派生类里的函数， 但是”对象.method()”在运行时始终调用该对象的函数。

如果想让对象也具有动态性（运行时确定将调用的函数）怎么办， 方法其实也是新壶装老酒， 就是用一个类去封装一个指针，**从而获得多态性**

为什么不直接使用指针，而非要让对象具有动态性呢？ 原因是如果使用指针， 那么必须确保指针指向的内存在使用指针之前没有被释放掉，否则就是使用悬垂指针。而当指针指向的动态内存时， 悬垂发生概率比较高， 因为程序员自己要操心指针使用时，该动态内存不能被delete/free。句柄类在初始化时， 它的初始化参数用的不是地址而是引用。且此对引用形参里的数据成员：指针，进行的复制不是简单的位拷贝， 而是申请一块该指针指向的对象的大小的内存， 然后使用该指针指向的对象来初始化这个内存。 这样即使引用形参里的指针指向的内存先于句柄类被释放， 也没有问题，因为对指针不是简单的位拷贝而是新申请内存，然后对该内存进行值初始化。**这样避免指针的劣势，无需决定何时对指针进行释放。**

句柄类的复制构造函数不同于其“默认“初始化构造函数， 对默认复制构造函数的形参（另一句柄类对象）所封装的指针采用简单的位初始化。

**句柄类里面的int \*use及复制控制成员，实现指针的智能性。**

见+~/cpp/class/handleClass.cpp

//handle class to wrap pointer to Item\_base

class handle {

public:

handle(const Item\_base &item);

handle(const handle &orig) {

m\_p = orig.m\_p;

p\_use = orig.p\_use;

++\*p\_use;

}

handle& operator=(handle &other);

~handle() {

if (0 == --\*p\_use) { delete m\_p; delete p\_use; }

}

/\*

const Item\_base& operator\*() {

return \*m\_p;

}

\*/

Item\_base\* operator->() {

return m\_p;

}

private:

Item\_base \*m\_p

int \*p\_use;

};

handle::handle(const Item\_base &item) {

m\_p = item.clone();

p\_use = new int(1); //"\*p\_use = 1" is wrong since we can initialize null pointer.

}

handle& handle::operator=(handle &other) {

++\*other.p\_use;

if (0 == --\*p\_use) {

delete m\_p;

delete p\_use;

}

m\_p = other.m\_p;

p\_use = other.p\_use;

return \*this;

}

## 通用句柄类

上面的句柄类handle具备3个功能， 能不能把其中1个或者2个功能给提取封装起来， 这样当某个类也想具有同样的功能时，它只用使用这个封装体就行，而不必在本类里面将3个功能同时实现。

通用句柄类存在目的就在此。

如果它封装了智能，那么使用者无需定义自己的复制控制成员。

如果它封装了指针， 虽然这个指针类型是基类，但只要指针实际可以指向派生类，多态性就具有 。所以多态性是可能的，depends on 这个指针是怎么被初始化的。

如果它能封装 “避免指针劣势“的这个功能，使用者就无须操心指针的释放。方案如下：

Template <typename T>

Class Handle<T> {

Public:

Handler(const T& item): p\_item(item.clone(), use(new int(1)) {} //这个具有多态性，但item必须有clone函数

Handler(const T& item): p\_item(new(item)), use(new int(1)) {} //这个虽然无需操心指针释放，但将丧失多态性的可能

Private:

T\* p\_item;

Int\* use;

}

但是否这样做，取决于需求。

如果需要这3个功能且item对象有clone函数，那么用第一个构造函数。

如果item是单一类，无派生，无需多态性，用第二个构造函数。

如果保证智能性，让用户决定多态性和指针释放，那么让用户类（封装handler的类）

自己决定怎样去间接初始化p\_item.

Class Handler {

Public:

Handler(T& p): p\_item(p), use(new int(1)) {} //这个具

Private:

T\* p\_item;

Int\* use;

}

Class Handler\_user {

Handler\_user(const T& item): h(new T(item) ) {} //无需担心你指针释放

Handler\_user(const T& item): handler(item.clone() ) {} //有多态性。

Private:

Handler<T> h;

}

见+~/cpp/class/handleClass.cpp ， General\_handler封装的是3个功能。

## Int &array\_ref[100] 是 array to reference,是不合法的. int (&array\_ref)[100] 是reference to array.

## 模板中， 不能定义与非类型形参同名的参数/函数名等

#include <iostream>

using namespace std;

template <int N, typename T>

//void foo(T v, int N) { //N use the same name of non-type parameter. wrong

void foo(T v, int i) {

cout << v << endl;

cout << N << endl;

cout << i << endl;

}

int main() {

// foo('c', 2); //wrong, cannot deduce N

foo<3> ('c', 2);

}

## 模板中，非类型形参的允许的类型

A non-type template-parameter shall have one of the following (optionally cv-qualified) types:

* Const integral or enumeration type,
* pointer to object or pointer to function,
* lvalue reference to object or lvalue reference to function,
* pointer to member function or pointer to member data.
* std::nullptr\_t

#include <iostream>

using namespace std;

template <int N, typename T>

void foo(T v) {

cout << v << endl;

cout << N << endl;

}

/\*

template <double D>

void foo2(void) {

NULL;

}

\*/

int main() {

foo<2>('c');

/\* wrong, i is not const

int i = 3;

foo<i>('d');

\*/

const int j = 4;

foo<j>('e');

/\* wrong, k is not integer or enum

const double k = 5.24;

foo<k>('f');

\*/

/\* wrong, â€˜doubleâ€™ is not a valid type for a template constant parameter

const double d = 3.14;

foo2<d>();

\*/

}

## x++ 是右值 in both c and C++.

## 前自增

In C, ++x is equivalent to x += 1。

In C++, ++x is equivalent to x += 1 if x is not a bool。

In c, 赋值操作符返回的是一个右值，所以无法对其赋值。

*An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment, but is not an lvalue.*

In C++, 赋值操作符返回的是一个赋值操作符左操作数的引用，为左值，所以可以对其赋值。

*The assignment operator (=) and the compound assignment operators all group right-to-left. All require a modifiable lvalue as their left operand and return an lvalue referring to the left operand.*

## 基类函数可以调用派生类里的函数。 甚至有些纯虚函数就是为了派生类重写，然后虚基类里面定义一个函数调用这个纯虚函数。

-bash-4.1$ cat can\_baseClass\_call\_methodOfDerived.cpp

#include <iostream>

using namespace std;

class Base {

public:

Base() { f(); }

void f() {

f\_toBeCall();

// f\_toBeCall\_nonVitrual(); //obviously, can't call a non-virtual function which is defined in derived class since base class don't know it exist.

}

virtual void f\_toBeCall()

{

cout << "f\_toBeCall in base class" << endl;

}

};

class Derived : public Base {

public:

void f\_toBeCall() {

cout << "f\_toBeCall in derived class" << endl;

}

void f\_toBeCall\_nonVirtual() {

cout << "f\_toBeCall\_nonVirtual in derived class" << endl;

}

};

class pureV\_base {

public:

void f() { //f() call function defined in derived class.

f\_toBeCall();

}

virtual void f\_toBeCall() = 0; //define pure virtual function here to let derived class rewrite it.

};

class pureV\_Derived : public pureV\_base {

public:

void f\_toBeCall() {

cout << "I am in derived class" << endl;

}

};

int main() {

Base B;

B.f();

Derived D;

D.f(); // the f() inherited from base call f\_toBeCall() & f\_toBeCall2() from derived class.

pureV\_Derived DV;

DV.f();

};

输出

-bash-4.1$ ./can\_baseClass\_call\_methodOfDerived.o

f\_toBeCall in base class

f\_toBeCall in base class

f\_toBeCall in base class

f\_toBeCall in derived class

I am in derived class

## 函数模板也可以重载， 可以使用不同参数的同名模板或者普通函数进行重载。 且选择可行函数的时候， 如果参数吻合， 普通函数具有最高优先级。

## Fync()

#### NAME

fsync - synchronise changes to a file

#### SYNOPSIS

#include <[unistd.h](http://pubs.opengroup.org/onlinepubs/7908799/xsh/unistd.h.html)>

int fsync(int *fildes*);

#### DESCRIPTION

The *fsync()* function can be used by an application to indicate that all data for the open file description named by *fildes* is to be transferred to the storage device associated with the file described by *fildes* in an implementation-dependent manner. The *fsync()* function does not return until the system has completed that action or until an error is detected.

The *fsync()* function forces all currently queued I/O operations associated with the file indicated by file descriptor *fildes* to the synchronised I/O completion state. All I/O operations are completed as defined for synchronised I/O file integrity completion.

#### RETURN VALUE

Upon successful completion, *fsync()* returns 0. Otherwise, -1 is returned and *errno* is set to indicate the error. If the *fsync()* function fails, outstanding I/O operations are not guaranteed to have been completed.

#### ERRORS

The *fsync()* function will fail if:

[EBADF]

The *fildes* argument is not a valid descriptor.

[EINTR]

The *fsync()* function was interrupted by a signal.

[EINVAL]

The *fildes* argument does not refer to a file on which this operation is possible.

[EIO]

An I/O error occurred while reading from or writing to the file system.

In the event that any of the queued I/O operations fail, *fsync()* returns the error conditions defined for [*read()*](http://pubs.opengroup.org/onlinepubs/7908799/xsh/read.html) and [*write()*](http://pubs.opengroup.org/onlinepubs/7908799/xsh/write.html).

#### EXAMPLES

None.

#### APPLICATION USAGE

The *fsync()* function should be used by programs which require modifications to a file to be completed before continuing; for example, a program which contains a simple transaction facility might use it to ensure that all modifications to a file or files caused by a transaction are recorded.

## The C++ compilation process

G++的一些编译option

* **-g** - turn on debugging (so GDB gives more friendly output)
* **-Wall** - turns on most warnings
* **-O** or **-O2** - turn on optimizations
* **-o <name>** - name of the output file
* **-c** – 停止link， output an object file (.o)
* **-I<include path>** - specify an include directory
* **-L<library path>** - specify a lib directory
* **-l<library>** - link with library lib<library>.a

## Exit(-1) 在c的 stdlib.h里面

## SOCKET相关

### 用于建立socket的sockaddr必须初始化

用memset(&addr, 0 sizeof addr)

或者 bzero(&addr, sizeof addr)/string.h 中

### Netstat –nautp

-n don’t resolve names

-a all

-u udp

-t tcp

-p display PID/Program for sockets

### 如果用inet\_pton去设置sockaddr的值的话， sockaddr里面的sin\_family必须设置。

否则会错误。 但是如果确定是IPV4的地址，用inet\_ptoa的话sin\_family可以不初始化，默认为AF\_INET.

sockaddr\_in srvAddr,;

bzero(&srvAddr, sizeof srvAddr);

inet\_pton(AF\_INET, SRVIP, &srvAddr.sin\_addr);

srvAddr.sin\_port = htons(SRVPORT);

srvAddr.sin\_family = AF\_INET;

### Addrinfo结构体

struct addrinfo {

int ai\_flags; // This field specifies additional options.

int ai\_family; // IPv4, IPv6 or IP agnostic.

int ai\_socktype; // TCP or UDP.

int ai\_protocol; // The protocol for the returned socket addresses.

size\_t ai\_addrlen; // Size of ai\_addr in bytes

struct sockaddr \*ai\_addr; // Containing the IP address and port.

char \*ai\_canonname; // The Canonical hostname.

struct addrinfo \*ai\_next; // linked list, next address.

};

### Sockaddr &sockaddr\_in

sockaddr is a generic descriptor for any kind of socket operation, whereas sockaddr\_in is a struct specific to IP-based communication (IIRC, "in" stands for "InterNet"). As far as I know, this is a kind of "polymorphism" : the bind() function pretends to take a struct sockaddr \*, but in fact, it will assume that the appropriate type of structure is passed in; i. e. one that corresponds to the type of socket you give it as the first argument.

sockaddr\_in and sockaddr\_in6 are both structures where first member is a sockaddr structure. According to the C standard, the address of a structure and its first member are the same, so you can cast the pointer to sockaddr\_in(6) in a pointer to sockaddr.

The sockaddr structure and sockaddr\_in structures below are used with IPv4. Other protocols use similar structures.

struct sockaddr {

ushort sa\_family;

char sa\_data[14];

};

struct sockaddr\_in {

short sin\_family;

u\_short sin\_port;

struct in\_addr sin\_addr;

char sin\_zero[8];

};

The sockaddr\_in6 and sockaddr\_in6\_old structures below are used with IPv6.

struct sockaddr\_in6 {

short sin6\_family;

u\_short sin6\_port;

u\_long sin6\_flowinfo;

struct in6\_addr sin6\_addr;

u\_long sin6\_scope\_id;

};

typedef struct sockaddr\_in6 SOCKADDR\_IN6;

typedef struct sockaddr\_in6 \*PSOCKADDR\_IN6;

typedef struct sockaddr\_in6 FAR \*LPSOCKADDR\_IN6;

struct sockaddr\_in6\_old {

short sin6\_family;

u\_short sin6\_port;

u\_long sin6\_flowinfo;

struct in6\_addr sin6\_addr;

};

sin\_addr底下还有一个：

antelope.sin\_addr.s\_addr = inet\_addr("10.0.0.1");

ipv6时为s6\_addr.

### Socket address family

/\*

\* Address families.

\*/

#define AF\_UNSPEC 0 /\* unspecified \*/

#define AF\_LOCAL 1 /\* local to host (pipes, portals) \*/

#define AF\_UNIX AF\_LOCAL /\* backward compatibility \*/

#define AF\_INET 2 /\* internetwork: UDP, TCP, etc. \*/

#define AF\_IMPLINK 3 /\* arpanet imp addresses \*/

#define AF\_PUP 4 /\* pup protocols: e.g. BSP \*/

#define AF\_CHAOS 5 /\* mit CHAOS protocols \*/

#define AF\_NS 6 /\* XEROX NS protocols \*/

#define AF\_ISO 7 /\* ISO protocols \*/

#define AF\_OSI AF\_ISO

#define AF\_ECMA 8 /\* European computer manufacturers \*/

#define AF\_DATAKIT 9 /\* datakit protocols \*/

#define AF\_CCITT 10 /\* CCITT protocols, X.25 etc \*/

#define AF\_SNA 11 /\* IBM SNA \*/

#define AF\_DECnet 12 /\* DECnet \*/

#define AF\_DLI 13 /\* DEC Direct data link interface \*/

#define AF\_LAT 14 /\* LAT \*/

#define AF\_HYLINK 15 /\* NSC Hyperchannel \*/

#define AF\_APPLETALK 16 /\* Apple Talk \*/

#define AF\_ROUTE 17 /\* Internal Routing Protocol \*/

#define AF\_LINK 18 /\* Link layer interface \*/

#define pseudo\_AF\_XTP 19 /\* eXpress Transfer Protocol (no AF) \*/

#define AF\_COIP 20 /\* connection-oriented IP, aka ST II \*/

#define AF\_CNT 21 /\* Computer Network Technology \*/

#define pseudo\_AF\_RTIP 22 /\* Help Identify RTIP packets \*/

#define AF\_IPX 23 /\* Novell Internet Protocol \*/

#define AF\_SIP 24 /\* Simple Internet Protocol \*/

#define pseudo\_AF\_PIP 25 /\* Help Identify PIP packets \*/

#define AF\_ISDN 26 /\* Integrated Services Digital Network\*/

#define AF\_E164 AF\_ISDN /\* CCITT E.164 recommendation \*/

#define pseudo\_AF\_KEY 27 /\* Internal key-management function \*/

#define AF\_INET6 28 /\* IPv6 \*/

#define AF\_NATM 29 /\* native ATM access \*/

#define AF\_ATM 30 /\* ATM \*/

#define pseudo\_AF\_HDRCMPLT 31 /\* Used by BPF to not rewrite headers

\* in interface output routine

\*/

#define AF\_NETGRAPH 32 /\* Netgraph sockets \*/

#define AF\_SLOW 33 /\* 802.3ad slow protocol \*/

#define AF\_SCLUSTER 34 /\* Sitara cluster protocol \*/

#define AF\_ARP 35

#define AF\_BLUETOOTH 36 /\* Bluetooth sockets \*/

#define AF\_MAX 37

### poll() 获取socket中的事件， 返回值0表示超时， -1表示错误， n表示有多少个socket成功获得。 本函数调用结束后， 将设置ufds的成员（struct）的revent值，可能值为POLLIN, POLLOUT, POLLPRI, POLLERR, POLLHUP,POLLHVAL.

Test for events on multiple sockets simultaneously

#### Prototypes

#include <sys/poll.h>

int poll(struct pollfd \**ufds*, unsigned int *nfds*, int *timeout*);

#### Description

This function is very similar to **select()** in that they both watch sets of file descriptors for events, such as incoming data ready to **recv()**, socket ready to **send()** data to, out-of-band data ready to **recv()**, errors, etc.

The basic idea is that you pass an array of struct pollfds in *ufds*, along with a timeout in milliseconds (1000 milliseconds in a second.) The *timeout* can be negative if you want to wait forever. If no event happens on any of the socket descriptors by the timeout, **poll()** will return.

Each element in the array of struct pollfds represents one socket descriptor, and contains the following fields:

struct pollfd {

int fd; // the socket descriptor

short events; // bitmap of events we're interested in

short revents; // when poll() returns, bitmap of events that occurred

};

Before calling **poll()**, load *fd* with the socket descriptor (if you set *fd* to a negative number, this struct pollfd is ignored and its *revents* field is set to zero) and then construct the *events* field by bitwise-ORing the following macros:

|  |  |
| --- | --- |
| POLLIN | Alert me when data is ready to **recv()** on this socket. |
| POLLOUT | Alert me when I can **send()** data to this socket without blocking. |
| POLLPRI | Alert me when out-of-band data is ready to **recv()** on this socket. |

Once the **poll()** call returns, the *revents* field will be constructed as a bitwise-OR of the above fields, telling you which descriptors actually have had that event occur. Additionally, these other fields might be present:

|  |  |
| --- | --- |
| POLLERR | An error has occurred on this socket. |
| POLLHUP | The remote side of the connection hung up. |
| POLLNVAL | Something was wrong with the socket descriptor *fd*—maybe it's uninitialized? |

#### Return Value

Returns the number of elements in the *ufds* array that have had event occur on them; this can be zero if the timeout occurred. Also returns -1 on error (and **errno** will be set accordingly.)

#### Example

int s1, s2;

int rv;

char buf1[256], buf2[256];

struct pollfd ufds[2];

s1 = socket(PF\_INET, SOCK\_STREAM, 0);

s2 = socket(PF\_INET, SOCK\_STREAM, 0);

// pretend we've connected both to a server at this point

//connect(s1, ...)...

//connect(s2, ...)...

// set up the array of file descriptors.

//

// in this example, we want to know when there's normal or out-of-band

// data ready to be recv()'d...

ufds[0].fd = s1;

ufds[0].events = POLLIN | POLLPRI; // check for normal or out-of-band

ufds[1] = s2;

ufds[1].events = POLLIN; // check for just normal data

// wait for events on the sockets, 3.5 second timeout

rv = poll(ufds, 2, 3500);

if (rv == -1) {

perror("poll"); // error occurred in poll()

} else if (rv == 0) {

printf("Timeout occurred! No data after 3.5 seconds.\n");

} else {

// check for events on s1:

if (ufds[0].revents & POLLIN) {

recv(s1, buf1, sizeof buf1, 0); // receive normal data

}

if (ufds[0].revents & POLLPRI) {

recv(s1, buf1, sizeof buf1, MSG\_OOB); // out-of-band data

}

// check for events on s2:

if (ufds[1].revents & POLLIN) {

recv(s1, buf2, sizeof buf2, 0);

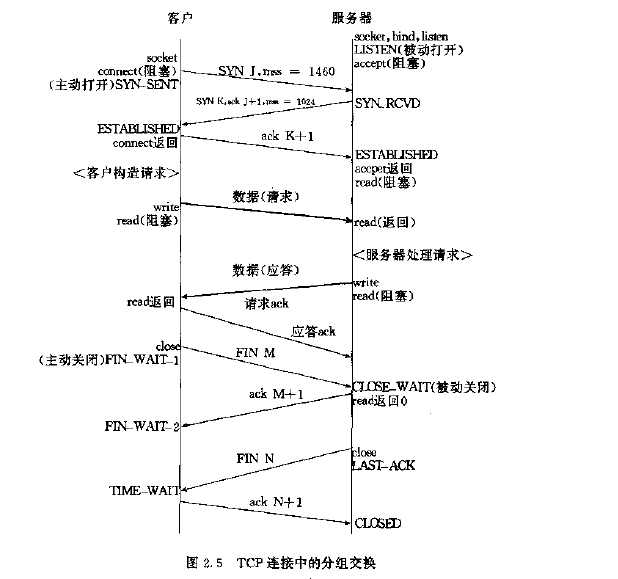
}

}

### Client Connect成功于server accept之前

这种情况是可能出现的， 只要connect request已经位于server request queue里， connect就能返回， server段accept与否取决于OS。

### TCP 建立和释放过程



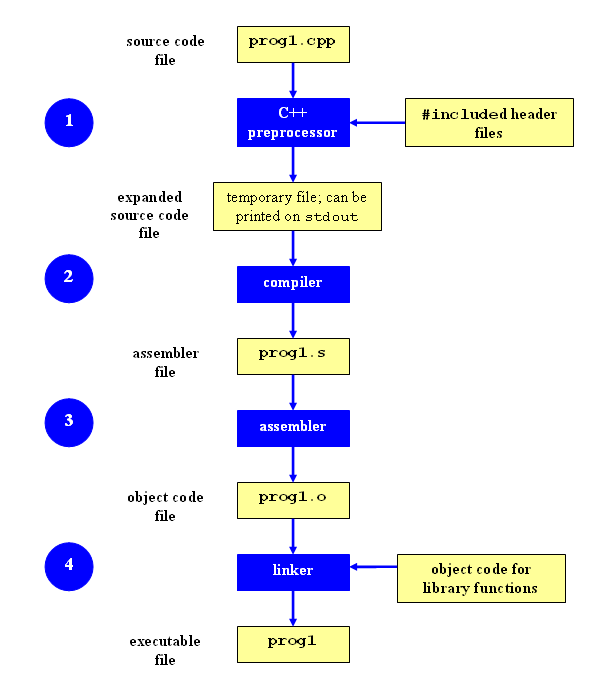
### 一个ip:port允许拥有多个socket, 例如某个网站使用以80端口接收。到底允许有多少个socket取决于listen的backlog值大小。

## Compiling a source code file in C++ is a four-step process.

For example, if you have a C++ source code file named prog1.cpp and you execute the compile command

g++ -Wall -ansi -o prog1 prog1.cpp

the compilation process looks like this:



1. The C++ preprocessor copies the contents of the included header files into the source code file, generates macro code, and replaces symbolic constants defined using #define with their values.
2. The expanded source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.
3. The assembler code generated by the compiler is assembled into the object code for the platform.
4. The object code file generated by the assembler is linked together with the object code files for any library functions used to produce an executable file.

By using appropriate compiler options, we can stop this process at any stage.

1. To stop the process after the preprocessor step, you can use the -E option:
2. g++ -E prog1.cpp

The expanded source code file will be printed on standard output (the screen by default); you can redirect the output to a file if you wish. Note that the expanded source code file is often incredibly large - a 20 line source code file can easily produce an expanded file of 20,000 lines or more, depending on which header files were included.

1. To stop the process after the compile step, you can use the -S option:
2. g++ -Wall -ansi -S prog1.cpp

By default, the assembler code for a source file named filename.cpp will be placed in a file named filename.s.

1. To stop the process after the assembly step, you can use the -c option:
2. g++ -Wall -ansi -c prog1.cpp

By default, the assembler code for a source file named filename.cpp will be placed in a file named filename.o.

## Static\_cast/dynamic\_cast

**Static Cast**

static\_cast doesn't do any run time checking of the types involved, which means that unless you know what you are doing, they could be very unsafe. It also only allows casting between related types, such as pointers or references between Base and Derived, or between fundamental types, such as long to int or int to float.

It does not allow casts between fundamentally different types（这里的fundamentally different type也包括如int\*->char \*这样类型）, such as a cast between a BaseA and BaseB if they are not related. This will result in a compile time error.

**Dynamic Cast**

dynamic\_cast is used for cases where you don't know what the dynamic type of the object is. You cannot use dynamic\_cast if you downcast and the argument type is not polymorphic. An example:

if(JumpStm \*j = dynamic\_cast<JumpStm\*>(&stm)) {

...

} else if(ExprStm \*e = dynamic\_cast<ExprStm\*>(&stm)) {

...

}

dynamic\_cast returns a null pointer if the object referred to doesn't contain the type casted to as a base class (when you cast to a reference, a bad\_cast exception is thrown in that case).

The following code is not valid, because Base is not polymorphic (it doesn't contain a virtual function):

struct Base { };

struct Derived : Base { };

int main() {

Derived d; Base \*b = &d;

dynamic\_cast<Derived\*>(b); // Invalid

}

An "up-cast" is always valid with both static\_cast and dynamic\_cast, and also without any cast, as an "up-cast" is an implicit conversion.

## Fork后的子进程并不能改写父进程里面相同变量的值

#include <unistd.h>

#include <stdio.h>

#include <unistd.h>

int main(){

int i = 0;

printf("Let's start\n");

pid\_t child = fork();

if (!child) {

i++;

}

usleep(1000); //sleep 1 s to let father process wait till "i++" executed in child process

if (!child) printf("Father: i is %d after sleep, the address of i is %p\n", i, &i); //i is 0

else printf("Child: i is %d after sleep, the address of i is %p\n", i, &i); // i is 1

//but curious thing is that the address of i is the same between child process and father process

}

打印出来的i的地址相同的原因： 打印出来的i的地址是虚拟弟子而不是物理地址。子进程“继承“父进程的变量， 在fork时整个虚拟地址空间被复制， 但是虚拟地址空间所对应的物理内存却没有复制（这个时候父子进程中变量i的虚拟地址都指向同一个物理地址）。 等子进程虚拟空间被写的时候， 将给子进程重新分配物理内存来复制父进程的物理地址（此时父子进程虚拟地址一样，但是物理地址不同）， 这就是”写时复制“。

## Fork的用法

1 #include <unistd.h>

2 #include <stdio.h>

3 #include <unistd.h>

4

5 void output(pid\_t pid) {

6 if(pid != 0){

7 printf("I am parent process. My PID is %d, my parent PID is %d\n", getpid(), getppid());

8 }

9 else {

10 printf("I am child process. My PID is %d, my parent PID is %d\n", getpid(), getppid());

11 }

12 }

13

14 int main(){

15 int i = 0;

16 printf("I am 1st\n");

17 pid\_t child = fork();

18 printf("The address of child is %p\n", &child);

19 printf("child is %d\n", child);

20 printf("The address of i is %p\n", &i);

21 printf("I am the 2nd, i is %d\n", i++);

22 output(child);

23 if (!child) {

24 pid\_t grandChild = fork();

25 output(grandChild);

26 }

27 usleep(1000);

28 printf("2nd time: The address of child is %p\n", &child);

29 printf("2nd time: child is %d\n", child);

30

31 }

* 1. 子进程的的其实执行代码点在fork这一行。如上的17， 24行。
  2. 子进程拷贝父进程的“进程数据”不意味这子进程和父进程共享memory， 只有vfork才共享memory。
  3. Fork（）这行完毕后，当前的process和fork出来的process均从”写fork返回值”(如上17行红色部分)继续向下执行.

因为该当前进程有刚刚fork出来的子进程，那么认为它为parent进程, 返回子进程的PID。当前进程刚刚fork出来的子进程，因为它没有任何子进程，所以被认为为子进程， fork 返回0.



另一段代码：

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

void usage(void) {

printf("usage:\n launch.o sever/msrp\_server.out|client/msrp\_client.out $ip $startPort, $endPort, $file\_to\_sent\n");

}

int main(int argc, char\* argv[]) {

if (argc < 6) {

usage();

}

char \*interval = "100";

int startport = atoi(argv[3]);

int endport = atoi(argv[4]);

char char\_startport[30];

sprintf(char\_startport, "%d", startport);

char \*newargv[6] = { argv[1], argv[2], char\_startport, interval, argv[5], NULL};

char \*envp[] = { NULL };

while (startport < endport) {

pid\_t pid = fork();

if (pid >0) {

startport += 100;

sprintf(newargv[2], "%d", startport);

}

else if (pid == 0) {

for(int i=0; i<6; i++) {

printf("%s ", newargv[i]);

}

printf("\n");

execve(argv[1], newargv, envp);

break;

}

else {

printf("error happened during fork\n");

break;

}

}

}

## 由于头文件没加造成的编译时错误 /乍看不那么明显

|  |  |
| --- | --- |
| in6\_addr\* GetBindIP() { return NULL;} | 10:14 |
| 这样的函数编译能过？ | 10:14 |

|  |  |
| --- | --- |
| **Hao Huang:** |  |
| 当然能了 | 10:14 |

|  |  |
| --- | --- |
| **Hong Jun Yang:** |  |
| 不能 | 10:14 |
| rtptransmitter.h:269: error: ISO C++ forbids declaration of 'in6\_addr' with no type rtptransmitter.h:269: error: 'in6\_addr' declared as a 'virtual' field rtptransmitter.h:269: error: expected ';' before '\*' token rtptransmitter.h:271: error: expected ';' before 'protected' |  |

## -int和unsigned int类型值比较

-int如-32比较居然比unsigned int如32大。 依以前的理解，类型不同的两个数操作时， 是进行类型提升的， 这样unsigned int会转换为 int 32, 那样比较结果应该和意料中相同， 但是结果却相反， 我想可能将-int转换为很大的int了。

unsigned int c = 274607296;

unsigned int d = 274608256;

unsigned int Interval = 320;

cout << ((c - d) > Interval) << endl; //1

cout << ((int)(c - d) > Interval) << endl; //1，证明这行和上一行一样Interval并没有转换为int

cout << ((int)(c - d) > (int)Interval) << endl; //0， 这个有个不足是， 万一 Interval > maxInt，就会出现错误。 所以正确比较应该是 if ( (c-d) >=0) { return c-d > Interval;} else { return false;}

cout << (int)(c-d) << endl;//-960

# Java

## Java中下一个shell命令，并得到输出

String[] cmd2 = new String[]{"sh", "-c", "ps -ef|grep Sb"};

Process proc = Runtime.getRuntime().exec(cmd2);

BufferedReader stdInput = new BufferedReader(new InputStreamReader(proc.getInputStream()));

BufferedReader stdError = new BufferedReader(new InputStreamReader(proc.getErrorStream()));

System.out.println("--------------stdInput");

String s = null;

while ((s = stdInput.readLine()) != null) {

System.out.println(s);

}

System.out.println("---------------stdError");

while ((s = stdError.readLine()) != null) {

System.out.println(s);

}

## Marshal, unmarshal, XML bind

An XML data binder accomplishes this by automatically creating a mapping between elements of the [XML schema](http://en.wikipedia.org/wiki/XML_schema) of the document we wish to bind and [members](http://en.wikipedia.org/wiki/Instance_variable) of a [class](http://en.wikipedia.org/wiki/Class_%28computer_science%29) to be represented in memory.

When this process is applied to convert an XML document to an object, it is called [unmarshalling](http://en.wikipedia.org/wiki/Serialization). The reverse process, to serialize an object as XML, is called [marshalling](http://en.wikipedia.org/wiki/Marshalling_%28computer_science%29).

## Java选项

### -Xmnsize: intial and maximum size of heap for new operation.

Sets the initial and maximum size (in bytes) of the heap for the young generation (nursery). The young generation region of the heap is used for new objects.

You can use -XX:NewSize to set the initial size and -XX:MaxNewSize to set the maximum size.

-Xmn256m

-Xmn262144k

-Xmn268435456

### –Xmssize: intial hep size

Sets the initial size (in bytes) of the heap. This value must be a multiple of 1024 and greater than 1 MB. If you do not set this option, then the initial size will be set as the sum of the sizes allocated for the old generation and the young generation..

-Xms6291456

-Xms6144k

-Xms6m

### –Xmxsize equivalent to -XX:MaxHeapSize.

Specifies the maximum size (in bytes) of the memory allocation pool in bytes. This value must be a multiple of 1024 and greater than 2 MB. The default value is chosen at runtime based on system configuration. For server deployments, -Xms and -Xmx are often set to the same value.

### –Xprof: 调试用。

Profiles the running program and sends profiling data to standard output. This option is provided as a utility that is useful in program development and is not intended to be used in production systems.

### –XshowSetting:category 显示JAVA设置情况. Category有如下几个选项

All : 显示所有种类的设置， 为默认值。

Locale： 显示当前现场设置。

Properties: 显示系统属性相关的设置

vm: 显示VM的设置。

### –Xss*size设置线程的栈大小，默认值如下：*

* Linux/ARM (32-bit): 320 KB
* Linux/i386 (32-bit): 320 KB
* Linux/x64 (64-bit): 1024 KB
* OS X (64-bit): 1024 KB
* Oracle Solaris/i386 (32-bit): 320 KB
* Oracle Solaris/x64 (64-bit): 1024 KB
* Windows: depends on virtual memory
* -Xss1m
* -Xss1024k
* -Xss1048576

### -XX:NativeMemoryTracking=*mode: 跟踪native memory使用情况。Mode如下：*

**off**

Do not track JVM native memory usage. This is the default behavior if you do not specify the -XX:NativeMemoryTracking option.

**summary**

Only track memory usage by JVM subsystems, such as Java heap, class, code, and thread.

**detail**

In addition to tracking memory usage by JVM subsystems, track memory usage by individual CallSite, individual virtual memory region and its committed regions.

### –XX:OnError=String： 敲String所示的命令，当不可恢复的错误发生时。

-XX:OnError="gcore %p;dbx - %p" (the %p designates the current process)

### –XX:MaxPermSize ： permanent generation使用的内存最大值

-XX:MaxPermSize=512m

-XX:PermSize=256m

### –XX:PermSize: permanent generation 使用的内存最小值。

### -Xinternalversion

Displays more detailed JVM version information than the -version option, and then exits.

## Java memory的分类

|  |  |  |  |
| --- | --- | --- | --- |
| **Memory Space** | **Start-up arguments and tuning** | **Monitoring strategies** | **Description** |
| Java Heap | -Xmx (maximum Heap space)  -Xms (minimum Heap size)  EX:  -Xmx1024m  -Xms1024m | - verbose GC  - JMX API  - JConsole  - Other monitoring tools | The Java Heap is storing your primary Java program Class instances. |
| PermGen | -XX:MaxPermSize (maximum size)  -XX:PermSize  (minimum size)  EX:  -XX:MaxPermSize=512m  -XX:PermSize=256m | - verbose GC  - JMX API  - JConsole  - Other monitoring tools | The Java HotSpot VM permanent generation space is the JVM storage used mainly to store your Java Class objects such as names and method of the Classes, internal JVM objects and other JIT optimization related data. |
| Native Heap   (C-Heap) | Not configurable directly.  For a 32-bit VM, the C-Heap capacity = 4 Gig – Java Heap - PermGen  For a 64-bit VM, the C-Heap capacity = Physical server total RAM & virtual memory – Java Heap - PermGen | - Total process size check in Windows and Linux  - pmap command on Solaris & Linux  - svmon command on AIX | The C-Heap is storing objects such as MMAP file, other JVM and third party native code object |

## Java 的访问属性

Modifier | Class | Package | Subclass | World

————————————+———————+—————————+——————————+———————

public | y | y | y | y

————————————+———————+—————————+——————————+———————

protected | y | y | y | n

————————————+———————+—————————+——————————+———————

no modifier | y | y | n | n

————————————+———————+—————————+——————————+———————

private | y | n | n | n

y: accessible

n: not accessible

# XML

## SCHEMA中<xs: sequence>属于indicator， 表示后面的element 要依次出现

<xs:element name="person">  
   <xs:complexType>  
    <xs:sequence>  
      <xs:element name="firstname" type="xs:string"/>  
      <xs:element name="lastname" type="xs:string"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>

# sourceInsight 使用技巧

## view->relation window: 显示某个symbol的关系网。

邮件点击relation window->relation window properties: 根据symbol的类型，定义不同的关系查找方式。如下面3种方式：

The relationships fall into three general categories, listed from computationally

the fastest to slowest:

**• Contains** – show the contents of the current symbol. For example, the

members of a struct.

**• Calls** – show what other symbols are referred to by the current symbol.

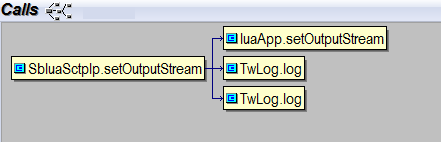
For example, functions that are called by the current function.

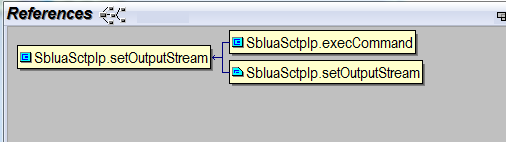
**• References** – show what other symbols refer to the current symbol.

For example, functions that call the current function.

例： 分析某个函数的关系网：

1. 点击这个函数， 显示relation windows， 方式选择calls。 目的：显示这个函数call哪些其他函数。



1. 在上面的windows里面， 右键new relation window->refresh relation window , 修改显示方式为references. 目的： 看这个函数被哪些其他函数调用。
2. 

## Relation window->右键选择show detail能够显示某个symbol下面的所有其他symbol

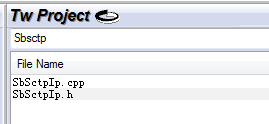
## 在project中查找某symbol

方法一： search->search project

方法二: look up reference.

连comment里面的内容都被查找了。可以使用 Search In来narrow查找范围

## 查找文件

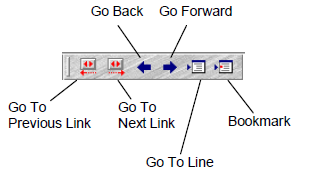


you type \*.c and press

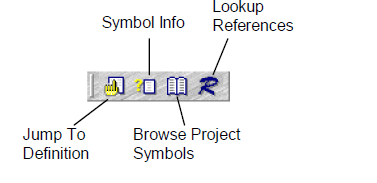
Enter, you will see all \*.c files in your project, regardless of directory. To remove

the wildcard, press \* (asterisk) and press Enter

## 设置标签



## 符号图标



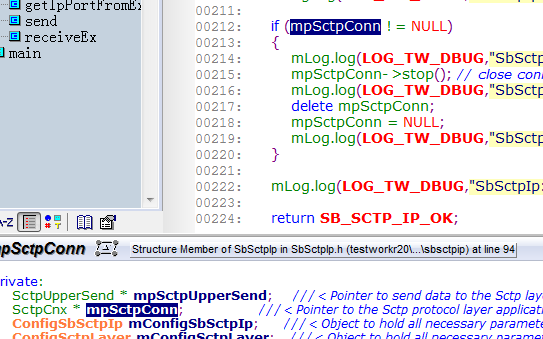
第一个是右键的快捷方式

第二个会给你讲这个symbol的info，是类还是对象什么的

第三个会显示当前project里面所有同名及相似的symbol

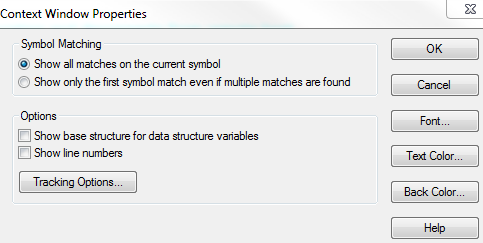
 最后一个会显示上一个search result。

## 在source file里面选择某个symbol后，Context window会自动查找这个symbol的base type



选定mpSctpConn后， 会显示SctpCnx的类型。 这不是我想要的， 关闭这个功能：

右键context window, 出现下图



去除show base structure for data structure variable

## Search result里面的东西是一个file buffer， 它的内容可以被编辑的。

# 调试技巧&SVN

## /var/log/message

这个文件记录系统里面的错误信息

## GDB调试

首先用g++ 带上-ggdb –Wall这个选项，生成可执行文件, 如g++ -ggdb –Wall –o main main.cc

然后gdb main进入gdb调试

然后run跑object文件

Backtrace看函数调用栈

X 地址 , 查看地址里面的值

Break 行/函数名， 设置断点

Condtion 1 变量==值， 为断点设置条件

Run 再次跑object文件

Step向下执行

Enter键， 可以重复上跳命令

P 变量名， 查看变量值。

在multiPorcess时，设置break后， 可以再设置set follow-fork-mode child，来使子进程在设定的breakpoint也break。 默认的follow-fork-mode 是parent的， break设置只对父进程起效

## 显示进程下线程

[root@localhost SOCKET]# ps -eT |grep 4422

4422 4422 pts/1 00:00:00 msrp\_server.out

4422 5642 pts/1 00:00:00 msrp\_server.out

## 显示正在运行的进程的backtrace

(gdb) attach 4422---4422是PID

(gdb) bt

#0 0x0000003ca8a0804d in pthread\_join () from /lib64/libpthread.so.0

#1 0x0000000000401a75 in main ()

(gdb) break TwLog.cpp:26 //早TwLog.cpp的26行break

## Strace –p pid && pstack pid

strace显示pid 的backtrace

pstack显示pid的stack

## 增加linux limit的方法

[tester@localhost ~]$ cat /etc/security/limits.d/90-nproc.conf

# Default limit for number of user's processes to prevent

# accidental fork bombs.

# See rhbz #432903 for reasoning.

\* soft nproc 65535

\* soft stack 2048

\* hard stack 2048

\* soft nofile 65535

\* hard nofile 65535

## 允许系统产生core dump文件

在文件/etc/security/limits.d/90-nproc.conf里面加上

**\*  soft  core  unlimited**

重启

**[root@localhost tester]# chkconfig abrtd off //关闭 automatic bug report service.**

[root@localhost tester]# echo 'coredump\_%e\_%p' > /proc/sys/kernel/core\_pattern //重写coredump pattern文件。

## SVN中conflict，merge

* 2个修改同一个文件，但是不在同一行， 不是conflict， SVN在check in时候能自动merge 2个人的change之处。 Update时候会把另一个人已经check in的change（如果这个人check in了） merge下来。
* 2个人修改了同一行，或者两个人重命名相同的文件， 会发生conflict， 此时如果check in/update， svn会提示

mc) mine-conflict - accept my version for all conflicts (same)   
(tc) theirs-conflict - accept their version for all conflicts (same)

(mf) mine-full - accept my version of entire file (even non-conflicts)  
(tf) theirs-full - accept their version of entire file (same)

F和C不同之处在于， full会discard其他人不同行的修改， 而conflict会执行正常的merge即其他人不同行的修改也会被合入。

## SVN 中HEAD 和BASE的区别

HEAD指最新版本。

BASE指你本地机器上的最新版本。