# C++

## Thread

代码见：<https://github.com/NexAlien/study/blob/master/cpp/laterCallbackToSpawnedThread/stdThread_version.cpp>

### Detach的作用：让子线程的执行独立于子线程object自身，即使子线程自身出了作用域，资源仍然不会释放，知道子线程执行完毕。 例子见<http://en.cppreference.com/w/cpp/thread/thread/detach>

### 注意但是如果子线程所在的main终止， 那所有资源将会被强制释放，除非在main最后使用pthread\_exit.

### 子thread被detach后就不能再join了，否则程序crash

### Join是等待子线程执行完毕才退出，在main函数里面调用pthread\_exit()是允许主线程退出，但是子线程资源不被释放。解释见<http://stackoverflow.com/questions/20824229/when-to-use-pthread-exit-and-when-to-use-pthread-join-in-linux>

## 如果一个类里面有引用成员变量，那么构造函数里面必须初始化这个成员变量。

## Static的成员变量在hpp文件声明后，只能在类外“定义”一次， 这个定义通常放到相应的cpp文件中

即使这个static的成员变量是private的，cpp文件里面也可以直接访问。见

<https://github.com/NexAlien/study/blob/master/cpp/basic/static/staticVariableDefinition/staticVariableDefinition.cpp>

## Static 成员函数无法改变非static 成员变量， 但static成员函数相当于全局函数

[hongjyan@euca-10-254-233-100 cpp]$ cat staticMemberFunctionChangeNormalMemberVariable.cpp

#include <string>

#include <iostream>

class Foo {

public:

//can not change non-static variable member via static member function。 原因是：将static函数是预分配的，相当于全局函数。而非static成员是在运行是建立instance时候附带创建的。前者先于后者存在。 所以static function不能访问非static成员。见下面的callStaticMethondOfPrivateConstructorClass.cpp

/\*

static std::string changeName(std::string& name) {

name\_ = name; //此行错误

return name\_; //此行错误

}

\*/

//因为foo作为参数传入了，所以这个static函数能够访问foo里面的non-static成员。

static std::string changeName(Foo& foo, const std::string& name) {

foo.name\_ = name;

return foo.name\_;

}

private:

std::string name\_;

};

int main() {

Foo foo;

//std::cout << foo.changeName("xiaoming") << std::endl;

std::cout << Foo::changeName(foo, "xiaoming") << std::endl;

return 0;

}

[hongjyan@euca-10-254-233-100 cpp]$ cat callStaticMethondOfPrivateConstructorClass.cpp

#include <string>

#include <iostream>

class Foo {

public:

static void sayHi() {

std::cout << "Hi" << std::endl;

}

private:

Foo(const std::string& name): name\_(name) {}

std::string name\_;

};

int main() {

Foo::sayHi();

return 0;

}

## Union长度为成员中最大的那个长度， 且union的内存可以重用的

struct ip6hdr {

union {

struct ip6hdrctl {

u\_int32\_t ctl6\_flow; /\* 24 bits of flow-ID \*/

u\_int16\_t ctl6\_plen; /\* payload length \*/

u\_int8\_t ctl6\_nxt; /\* next header \*/

u\_int8\_t ctl6\_hlim; /\* hop limit \*/

} un\_ctl6;

u\_int8\_t un\_vfc; /\* 4 bits version, 4 bits reserved \*/

} ip6\_ctlun;

struct in6\_addr ip6\_src; /\* source address \*/

struct in6\_addr ip6\_dst; /\* destination address \*/

};

如上union ip6\_ctlun内的成员， un\_vfc和un\_ctl6并不是非此即彼的， 而只是指向不一样大小和长度的地址所用。用户可以ip6\_ctlun.un\_ctl6.ctl6\_flow = 3; 然后ip6\_ctlun.un\_vfc = 6;

这个顺序是提醒用户，先赋值un\_ctl6,然后再赋值un\_vfc。 如果顺序反过来的话， un\_vfc内容将被un\_ctl6重写。

## Dup和dup2

int dup(int oldfd);

int dup2(int oldfd, int newfd);

dup2() makes newfd be the copy of oldfd（将newfd作为oldfd的拷贝，即原来对newfd的IO变成对oldfd的IO）, closing newfd first if necessary(似乎从来没有close newfd，至少通过代码实验dup2后， 那个newfd并没有被腾出来)。

而dup(int oldfd)相当于dup2(int oldfd, int lowest-numbered fd), 即第二个参数是系统可用的最小fd。

这2个函数的返回值都是那个newfd。

-bash-4.1$ cat ownPipe\_useDup.cpp

#include <stdio.h>

#include <cstdlib>

#include <unistd.h>

#include <sys/wait.h>

int main() {

int pfds[2];

pipe(pfds);

if (!fork()) {

close(1);

dup(pfds[1]);

execlp("ls", "ls", NULL);

} else {

close(0);

dup(pfds[0]); //将来所有的读STDIN的， 都从pfds[0]里读

close(pfds[1]); //close pfds[1] in case result would print to pfds[1] instead of stdout

execlp("wc", "wc", "-l", NULL); //wc –l 没有参数的话，将从STDIN读，即从pfds[0]里面读

}

wait(NULL);

return 0;

}

but note the following:

，

\* If oldfd is not a valid file descriptor, then the call fails, and newfd is not closed.

\* If oldfd is a valid file descriptor, and newfd has the same value as oldfd, then dup2() does nothing, and returns newfd.

以上代码同样可以用dup2来实现

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

int main() {

int pfds[2];

pipe(pfds);

printf("START---pfds[0] is %d, pfds[1] is %d\n", pfds[0], pfds[1]);

if (!fork()) { //child process

dup2(pfds[1], STDOUT\_FILENO);

execlp("ls", "ls", NULL);

}

else { //parent process

dup2(pfds[0], STDIN\_FILENO);

close(pfds[1]); //will print to pfds[1] instend of STDOUT if not close pfds[1]. Which cause ternimal always is in "

input state", reason unknown.

execlp("wc", "wc", "-l", NULL);

}

wait(NULL);

return 0;

}

## 父进程死亡时终结子进程

prctl(PR\_SET\_PDEATHSIG, SIGHUP);

## 外围类没有访问嵌套类private成员的权利。定义嵌套类只是相当于给嵌套类加了个namespace。 无论类定义在哪，访问标号起同样作用

#include <cstdio>

class Cat {

public:

public:

Cat() { \_catImpl = new CatImpl(); }

~Cat() { delete \_catImpl; }

void purr() {

\_catImpl->purr();

}

private:

class CatImpl {

friend class Cat; //必须声明这个。否则外围类不能访问嵌套类的purr(), 除非purr()定义在 public下。

void purr() {

printf("purrrr\n");

}

};

CatImpl\* \_catImpl;

};

int main() {

Cat cat;

cat.purr();

//Cat::CatImpl catImpl; //CatImpl定义下Cat的private成员下， 外界无法创建。

}

似乎和：定义一个独立私有CatImpl类。然后设置Cat 为CatImpl friend。所起的作用一样。

## Getopt

具体见<https://www.gnu.org/software/libc/manual/html_node/Getopt.html>

#include <unistd.h>

#include <cstdio>

#include <cstdlib>

#include <ctype.h>

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

int aflag = 0;

int bflag = 0;

char\* cvalue = NULL;

int index;

int c;

opterr = 1; //If an error was detected, and the first character of optstring is not a colon,

//and the external variable opterr is non-zero (which is the default), getopt() prints an error message.

while (-1 != (c = getopt(argc, argv, "abc:"))) { // : indicts that there is a argument following option:c

switch (c) {

case 'a':

aflag = 1;

break;

case 'b':

bflag = 1;

break;

case 'c':

cvalue = optarg; //optarg is extern global variable define in getopt library.

break;

case '?': //unknow option or option-without-argument

if (optopt == 'c') fprintf(stderr, "Option -%c requires an argument.\n", optopt);

else if (isprint(optopt)) fprintf(stderr, "Unknown option '-%c'.\n", optopt);

else fprintf(stderr, "Unknown option character'\\x%x'.\n", optopt);

return -1;

default:

exit(-1);

}

}

printf("aflag is %d, bflag is %d, cvalue is %s\n", aflag, bflag, cvalue);

//unknown argument will come here.

for (index = optind; index < argc; index++) //optind is the index of the first argv-element that is not an option.

printf("Non-option argument %s\n", argv[index]);

return 0;

}

## 不能在头文件中定义全局变量，而是声明， 否则会报重复定义的错误

example.h

int foo;

You would optionally define it on a translation unit:

example.c

int foo = 2;

And you would refer to it on some other(s) translation unit(s):

example-main.c

#include <stdio.h>

#include "example.h"

int main()

{

printf("foo = %d\n", foo);

return 0;

}

## Beej.IPC

<http://beej.us/guide/bgipc/output/html/singlepage/bgipc.html#audience>

### FILE\* 表明一个file descriptor， 它只是一个ints， 可以用fileno(FILE\*)获取文件描述符的整形值。

## 动态库的使用

使用dlopen， dlsym不需要include头文件了

-bash-4.1$ ls

Printf.cpp Printf.h makefile test.cpp

-bash-4.1$ cat Printf.h

#ifndef PRINTF

#define PRINTF

void Printf(void);

#endif

-bash-4.1$ cat Printf.cpp

#include <stdio.h>

extern "C" { //disable C++ mangled

void Printf(void) {

printf("I am Printf\n");

}

}

-bash-4.1$ cat test.cpp

//#include "Printf.h"

#include <dlfcn.h>

#include <stdio.h>

#include <stdlib.h>

void Printf() {

printf("I am Printf in test.cpp\n");

}

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

void\* lib\_handler = dlopen("libPrintf.so", RTLD\_LAZY|RTLD\_LOCAL);

if (!lib\_handler) {

printf("[%s] unable to load library: %s\n", \_\_FILE\_\_, dlerror());

exit(EXIT\_FAILURE);

}

void\* p = dlsym(lib\_handler, "Printf");

if (!p) {

printf("[%s] unable to get symbol: %s\n", \_\_FILE\_\_, dlerror());

exit(EXIT\_FAILURE);

}

void (\*Printf)() = (void (\*) (void))p;

Printf();

dlclose(lib\_handler);

}

-bash-4.1$ cat makefile

all: libPrintf test.o

test.o: test.cpp

g++ -Wall -ggdb -L. -lPrintf -ldl test.cpp -o test.o

libPrintf: Printf.cpp

g++ -fPIC -shared Printf.cpp -o libPrintf.so

clean:

rm -rf \*.o \*.so

-bash-4.1$

## Exec系列的第二个参数是第一个参数（程序）的别名

execl("/workspace/home/hongjyan/SS\_LibGetBindIP/SS\_LibGetBindIP/gbip/build/gbip", "gbip", "--interface=eth0", "--hasmoname=/SSH", "IP", NULL);

## 似乎不能在namespace/类/函数里面include标准库的头文件

## Static library和dynamic library之间的区别

Shared libraries are .so (or in Windows .dll, or in OS X .dylib) files. All the code relating to the library is in this file, and it is referenced by programs using it at run-time. A program using a shared library only makes reference to the code that it uses in the shared library. Shared library的使用者被run-time 引用code in the shared library。

Static libraries are .a (or in Windows .lib) files. All the code relating to the library is in this file, and it is directly linked into the program at compile time. A program using a static library takes copies of the code that it uses from the static library and makes it part of the program. [Windows also has .lib files which are used to reference .dll files, but they act the same way as the first one]. Static library的使用者会从static library里面拷贝将要使用的代码块， 作为自身程序的一部分。

There are advantages and disadvantages in each method.

Shared libraries reduce the amount of code that is duplicated in each program that makes use of the library, keeping the binaries small.//由于shared library的使用者只是在需要时引用库，而不是拷贝，所以使用者自身的程序大小不会因为使用库而增大。 It also allows you to replace the shared object with one that is functionally equivalent, but may have added performance benefits without needing to recompile the program that makes use of it. //甚至可以以提供相同函数功能的新库replace先前的静态库。Shared libraries will, however have a small additional cost for the execution of the functions as well as a run-time loading cost as all the symbols in the library need to be connected to the things they use. //缺点是使用这要实时load shared library。 Additionally, shared libraries can be loaded into an application at run-time, which is the general mechanism for implementing binary plug-in systems.

Static libraries increase the overall size of the binary, but it means that you don't need to carry along a copy of the library that is being used. As the code is connected at compile time there are not any additional run-time loading costs.//静态库的使用者在编译期间就copy了静态库里面使用代码了， 编译好了， 代码就在那里了。 The code is simply there.

Personally, I prefer shared libraries, but use static libraries when needing to ensure that the binary does not have many external dependencies that may be difficult to meet, such as specific versions of the C++ standard library or specific versions of the Boost C++ library.//当库对外部依赖太多， 如需要特定的C++标准库或者Boost库时， 优先使用静态库。 因为静态库把所依赖的都包含在那里了。

## Library 的source compatible和binary compatible

A library is **binary compatible**, if a program linked dynamically to a former version of the library continues running with newer versions of the library without the need to recompile. 使用升级后的library后，用户代码不需要重新编译， 这样的library称为binary compatible

If a program needs to be recompiled to run with a new version of library but doesn't require any further modifications, the library is **source compatible**. 使用升级后的library后， 用户代码不需要修改，但需要重新编译， 这样的library称为source compatible。

## fgets读完最后一行后, feof并不会为true, 再次调用一次fgets后feof才会为true

while (!feof(rfd)) {

fgets(line, 999, rfd);

/\*dealing line

printf("%s", line);

\*/

memset(line, 0, 1000); //此处加上此行, 读和处理line后， 置0 line。使读取最后一行后, fgets 再读一遍停留在 EOF, line内容为空，没东西处理。

}

## 注意struct的4字节对齐。

struct st\_ageWeight {

int age;

char c;

int weight;

};

printf("sizeof(st\_ageWeight) is %u\n", sizeof(st\_ageWeight));

st\_ageWeight st\_person = {21, 'c', 0x45};

struct st\_ageWeight \*p\_st\_person = &st\_person;

char \*\*pp\_st\_person = (char\*\*)&p\_st\_person;

int age = readIntFromBuf(pp\_st\_person);

char m = \*\*pp\_st\_person;

char m\_next = \*(\*pp\_st\_person+1);

char m\_next\_next = \*(\*pp\_st\_person+2);

(\*pp\_st\_person)++;

int weight = readIntFromBuf(pp\_st\_person);

printf("age is %d, c is %c, m\_next is %x, m\_next\_next is %x, weight is %x\n", age, m, m\_next, m\_next\_next, weight);

输出为：sizeof(st\_ageWeight) is 12

age is 21, c is c, m\_next is 7f, m\_next\_next is 0, weight is 4500007f

所以可以看出来， 加的字节是在char c之后加入的。

## 不能使用&&取2重指针

int I = 0;

int \*\*pp = &&I; //不能编译

原因是\*\*pp是指针指向指针\*p, 但\*p在内存中并不存在，虽然&i地址是OK的。要分2步走

Int I = 0;

Int \*p = &i；

Int \*\*pp = &p;

## 负数表示

N-bit表示的数字-x的负数 为正数2^N – x在内存中的表现， 代码过程如下：

1 = 0000 0001

~1 = 1111 1110

~1 + 1 = 1111 1111 //注意这里是+1

-1 = 1111 1111

21 = 0001 0101

~21 = 1110 1010

~21+1 = 1110 1011

-21 = 1110 1011

下面是一个容易出错的例子：

char temp[200]; //正确结果要改为unsigned char

memset(temp, 0, 200);

temp[0] = 0xfe; //因为char，所以是有符号类型的，而内存中bit位为11111110(11111110-1 = 11111101🡪 ~11111101🡪00000010,故此值为-2.

int itemp = temp[0];

int test = 0xfe; //这个正确原因：0xfe代表值，而非内存内容，写到内存前msb为0.

printf(“itemp is %x, %d\n”, itemp, itemp); //0xfffffffe, -2， %x表示已hex打印unsigned integer， 所以将一字节-2转换为4字节-2，即0xfffffffe.

printf("temp[0] is %02x\n", temp[0]); //0xfffffffe，02也没用，当0表示padding，2表示位数，当位数比实际数位小时，并不会truncated

printf("temp[0] is %x\n", temp[0]); //0xfffffffe

printf("test is %x\n", test); //0xfe

红色输出非我们

## #的作用

**#： 如果在宏前面加上#，表明stringify，如：**

#define WARN\_IF(EXP) \

do { if (EXP) \

fprintf (stderr, "Warning: " #EXP "\n"); } \

while (0)

WARN\_IF (x == 0);

==> do { if (x == 0)

fprintf (stderr, "Warning: " "x == 0" "\n"); } while (0);

**宏参数不能被stringify：**

#define foo 4

#define str(s) #s

Str(foo)

🡺 “foo”

**此时可以在str宏外面包一层宏， 逃避这个规律：**

#define xstr(s) str(s)

#define str(s) #s

#define foo 4

xstr(foo)

🡺xstr(4);

🡺str(4)

🡺”4”

原因如下： s is stringified when it is used in str, so it is not macro-expanded first. But s is an ordinary argument to xstr, so it is completely macro-expanded before xstr itself is expanded (see [Argument Prescan](https://gcc.gnu.org/onlinedocs/cpp/Argument-Prescan.html#Argument-Prescan)). Therefore, by the time str gets to its argument, it has already been macro-expanded.

英文解释如下：<https://gcc.gnu.org/onlinedocs/cpp/Stringification.html#Stringification>

## ##的作用：作为连接使用。 称“token pasting”或”token concatenation“. 在此操作符2边或一边， 有一个token为macro argument时有用，见下：

如

#define PLUS +

#define EQUAL =

#define PLUS\_EQU PLUS##EQUAL

否则#define PLUS\_EQU PLUSEQUAL是不行的。

比较真实的案例如下：

struct command

{

char \*name;

void (\*function) (void);

};

struct command commands[] =

{

{ "quit", quit\_command },

{ "help", help\_command },

...

};

可以写为

#define COMMAND(NAME) { #NAME, NAME ## \_command }

struct command commands[] =

{

COMMAND (quit),

COMMAND (help),

...

};

#和##的综合使用：

#include <stdio.h>

#define STRINGIFY2(x) #x

#define STRINGIFY(x) STRINGIFY2(x)

#define PASTE2(a, b) a##b

#define PASTE(a, b) PASTE2(a, b)

#define BAD\_PASTE(x,y) x##y

#define BAD\_STRINGIFY(x) #x

#define SOME\_MACRO function\_name

int main()

{

printf( "buggy results:\n");

printf( "%s\n", STRINGIFY(BAD\_PASTE(SOME\_MACRO, \_\_LINE\_\_)));

printf( "%s\n", BAD\_STRINGIFY( BAD\_PASTE( SOME\_MACRO, \_\_LINE\_\_)));

printf( "%s\n", BAD\_STRINGIFY( PASTE( SOME\_MACRO, \_\_LINE\_\_)));

printf( "\n" "desired result:\n");

printf( "%s\n", STRINGIFY(PASTE(SOME\_MACRO, \_\_LINE\_\_)));

}

输出如下：

buggy results:

SOME\_MACRO\_\_LINE\_\_

BAD\_PASTE( SOME\_MACRO, \_\_LINE\_\_)

PASTE( SOME\_MACRO, \_\_LINE\_\_)

desired result:

function\_name21

## 函数指针类型比较奇葩的出现

int\* (\*pFunc) (float a, char\* s);

pFunc = (int\* (\*) (float, char\*)) (void\_P);

## 函数指针和typedef

#include <stdio.h>

//Sigfunc is function. Sigfunc是一个函数类型

typedef void Sigfunc(int);

//Sigfunc\_t is a function pointer。 Sigfunc\_t 是一个函数指针类型

typedef void (\*Sigfunc\_t)(int);

void sig\_chld(int signo) {

printf("sig\_chld\n");

}

//Do not use typedef, looks mess.

//if a function return a "function pointer", \* must be adjacent to function name closely. 如果一个函数的返回值类型是函数指针， 那么这个函数类型为returntypeof\_returnfunc **(\***funcname(p1, p2, p3)**)** (returnfunc\_p1, returnfunc\_P2, returnfunc\_p3)。

void (\*signal(int signo, void (\*sig\_handler) (int))) (int) {

sig\_handler(0);

printf("sighandler address is %p\n", sig\_handler);

return sig\_handler;

}

//use typedef function

Sigfunc\* signal\_func(int signo, Sigfunc\* func) {

func(0);

printf("func address is %p\n", func);

return func;

}

//use typedef function pointer

Sigfunc\_t signal\_func\_t(int signo, Sigfunc\_t func\_t) {

func\_t(0);

printf("func\_t address is %p\n", func\_t);

return func\_t;

}

//return a "function pointer" which has not any parameter & return nothing. 比较奇葩的一个例子，returnfoo返回值类型为一个void （\*） ().

void foo() {

printf("foo\n");

}

void (\*returnfoo(int x)) () {

printf("returnfoo\n");

return foo;

}

//return a "function pointer" which has not any parameter & return a void\* pointer. 次奇葩的一个例子，returnbar返回值类型为一个void\* （\*） ().

void\* bar() {

printf("bar\n");

return NULL;

}

void\* (\*returnbar(int x)) () {

printf("returnbar\n");

return bar;

}

int main() {

Sigfunc \*pfunc; //must add \* before pfunc. 此处必须带\*, 因为Sigfunc被typedef为函数类型而非函数指针。有点奇怪的是函数名作为参数传递时自动转化为函数地址， 但是函数类型却不直接转化为函数指针。

Sigfunc\_t func\_t;

//c standard says that a function name in this context is converted to the address of function, so,

//it is ok to pass sig\_chld or &sig\_chld as the second parameter of signal, signal\_xx

pfunc = signal(0, sig\_chld);

func\_t = signal(0, sig\_chld);

printf("pfunc is %p\n", pfunc);

printf("func\_t is %p\n", func\_t);

pfunc = signal\_func(0, sig\_chld);

func\_t = signal\_func(0, sig\_chld);

printf("pfunc is %p\n", pfunc);

printf("func\_t is %p\n", func\_t);

pfunc = signal\_func\_t(0, sig\_chld);

func\_t = signal\_func\_t(0, sig\_chld);

printf("pfunc is %p\n", pfunc);

printf("func\_t is %p\n", func\_t);

returnfoo(0)(); //调用函数返回函数指针，然后给这个“返回的这个函数指针”参数以再次调用。

returnbar(0)();

}

## #define 和#typedef之间的区别

#define is a preprocessor token: the compiler itself will never see it.

#typedef is a compiler token: the preprocessor does not care about it.

## Size\_t 和ssize\_t的区别

Size\_t 是unsigned的。 ssize\_t可能是负数， 所以如果值可能为负的话要用ssize\_t。用size\_t/ssize\_t而不用int的目的是portable， 为啥还不知。

## New的隐式用法

标准函数如下：

|  |  |
| --- | --- |
| **throwing (1)** | void\* operator new[] (std::size\_t size) throw (std::bad\_alloc); |
| **nothrow (2)** | void\* operator new[] (std::size\_t size, const std::nothrow\_t& nothrow\_value) throw(); |
| **placement (3)** | void\* operator new[] (std::size\_t size, void\* ptr) throw(); |

而一般我们的用法：

New int[5], 实际上用了函数1， int[5]长度作为第一个参数size。 似乎”表达式最后面的长度”作为”第一个参数”. 官方解释如下：

operator new[] can be called explicitly as a regular function, but in C++, new[] is an operator with a very specific behavior: An expression with the new operator on an array type, first calls function operator new (i.e., this function) with the size of its array type specifier as first argument (plus any array overhead storage to keep track of the size, if any), and if this is successful, it then automatically initializes or constructs every object in the array (if needed). Finally, the expression evaluates as a pointer to the appropriate type pointing to the first element of the array.

网站例子如下：

*struct* MyClass {

*int* data;

MyClass() {std::cout << '\*';} *// print an asterisk for each construction*

};

*int* main () {

std::cout << "constructions (1): ";

*// allocates and constructs five objects:*

MyClass \* p1 = *new* MyClass[5];

std::cout << '\n';

std::cout << "constructions (2): ";

*// allocates and constructs five objects (nothrow):*

MyClass \* p2 = *new* (std::nothrow) MyClass[5];

std::cout << '\n';

*delete*[] p2;

*delete*[] p1;

*return* 0;

}

“Placement new”解释见： <https://isocpp.org/wiki/faq/dtors#placement-new>

## Iostream::ignore这个挺好用

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

istream& ignore (streamsize n = 1, int delim = EOF);

Extract and discard characters

Extracts characters from the input sequence and discards them, until either n characters have been extracted, or one compares equal to delim.

## 列出.a文件里面的.o文件, 显示.a文件里的符号表

ar t /usr/lib/libLiS.a|pr -4 –t

*nm -C /usr/lib/libstdc++-3-libc6.2-2-2.10.0.a //*

## Linux中不同路径下文件的用处

1. /usr/bin : for executables (and scripts)
2. /usr/lib : for binary libraries (static (.a) and shared libraries (.so))
3. /usr/include : for header files (all languages)

Yum install的东西会放到这些文件夹中

1. /usr/local/bin，
2. /usr/local/lib
3. /usr/local/include

同上面3个， 只不过似乎/usr/local/lib并不是默认路径

## Ldd

ldd /opt/testworks/bin/SbRtpUdpIp.bin.orig 查看某个binrary文件使用了哪些库

## 编译时搜索库文件的顺序

并不是按照$PATH中来的：

Unix systems tend to sort different types of files into different directories. There are separate directories for executables (e.g. /usr/bin), documentation (e.g. /usr/man, /usr/info, …), native libraries (e.g. /usr/lib), Perl libraries (e.g. /usr/lib/perl5), and so on. Correspondingly, there are different path variables, all with the same syntax (colon-separated list of directories): PATH for executables, MANPATH for man pages, LD\_LIBRARY\_PATH for native libraries¹, PERL5LIB for Perl libraries, and so on.

Order1：UNIX中搜索$LD\_LIBRARY\_PATH里的内容.🡪似乎建议不使用

Order2： rpath encoded in the binary?

Order3: 默认路径如：/lib, /lib64, /usr/lib, /usr/lib64， /usr/local/lib, /usr/local/lib64. Linux中还有/etc/ld.so.conf中的内容

所以如果想让自己的库加入系统库中，有下列做法：

1.将你自己的路径$DIR加入/etc/ld.so.conf中， 然后调用ldconfig让这个配置文件起效

2. 编译过程中export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:$DIR

3. 在默认路径如/usr/lib里面加入软连接到自己的库如：ln –s $DIR /usr/lib

参见[http://www.eyrie.org/~eagle/notes/rpath.html 来看另一个通过改变RPATH](http://www.eyrie.org/~eagle/notes/rpath.html%20%20%20%20%20%20%20来看另一个通过改变RPATH)的方式

## .a, .so, .la文件

.so files are **dynamic libraries**. The suffix stands for "shared object", because all the applications that are linked with the library use the same file, rather than making a copy in the resulting executable.

.a files are **static libraries**. The suffix stands for "archive", because they're actually just an archive (made with the ar command -- a predecessor of tar that's now just used for making libraries) of the original .o object files.

.la files are **static libraries used by the GNU "libtools" package**. You can find more information about them in this question: [What is libtool's .la file for?](http://stackoverflow.com/questions/1238035/what-is-libtools-la-file-for)

Static and dynamic libraries each have pros and cons.

Static pro: The user always uses the version of the library that you've tested with your application, so there shouldn't be any surprising compatibility problems.

Static con: If a problem is fixed in a library, you need to redistribute your application to take advantage of it. However, unless it's a library that users are likely to update on their own, you'd might need to do this anyway.

Dynamic pro: Your process's memory footprint is smaller, because the memory used for the library is amortized among all the processes using the library.

Dynamic pro: Libraries can be loaded on demand at run time; this is good for plugins, so you don't have to choose the plugins to be used when compiling and installing the software. New plugins can be added on the fly.

Dynamic libraries are especially useful for system libraries, like libc. These libraries often need to include code that's dependent on the specific OS and version, because kernel interfaces have changed. If you link a program with a static system library, it will only run on the version of the OS that this library version was written for. But if you use a dynamic library, it will automatically pick up the library that's installed on the system you run on.

## 写日志

void Printf(char\* filePath, char \*out, ...)

{

char printf\_buf[1024];

va\_list args;

int printed;

FILE \*fp;

if(filePath == NULL)

fp = fopen("/home/huanghao/Log.log","a+");

else

fp = fopen(filePath, "a+");

int length = -1;

if (fp != NULL) {

va\_start(args, out);

printed = vsprintf(printf\_buf, out, args);

va\_end(args);

fseek(fp,SEEK\_SET,SEEK\_END);

fputs(printf\_buf, fp);

fclose(fp);

}

return;

}

## Endian

单个byte中， 同一地址，高位在后面，低位在前面。 如0x41

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

代码如下：

typedef unsigned char uint8\_t;

typedef struct {

uint8\_t a:1;

uint8\_t b:1;

uint8\_t c:1;

uint8\_t d:1;

uint8\_t e:1;

uint8\_t f:1;

uint8\_t g:1;

uint8\_t h:1;

} stuc;

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

char c = 0x41;

printf("Low bit--->High bit\n");

for(int i=0; i<8; i++) {

printf("%d: bit is %d\n", i, (c>>i) & 0x01);

}

stuc \*st = (stuc \*) &c;

printf("st->a is %d\n", st->a);

printf("st->b is %d\n", st->b);

printf("st->c is %d\n", st->c);

printf("st->d is %d\n", st->d);

printf("st->e is %d\n", st->e);

printf("st->f is %d\n", st->f);

printf("st->g is %d\n", st->g);

printf("st->h is %d\n", st->h);

输出为

st->a is 1

st->b is 0

st->c is 0

st->d is 0

st->e is 0

st->f is 0

st->g is 1

st->h is 0。

假如内存像一张白纸， 白纸开始是低地址， 后来是高地址， 通常如果我们想写一个2字节的hex如b34f在白纸上， 我们会先写b3,再4f, 那么高位b3被先存于低地址：高位存在低地址，而bigEndian也是这么做的。 如int i = 0x81124110

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Addr:0x01 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Addr:0x02 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| Addr:0x03 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Addr:0x04 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

而X86架构的相反： 先写4f， 再写b3， 这样，低位存在低地址， 高位存在高地址。 这叫little endian。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Addr:0x01 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Addr:0x02 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Addr:0x03 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| Addr:0x04 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

X86中代码如下：

int i = 0x81124110；

char \*p = (char \*) &i;

printf("Low address--->High address\n");

for (int j=0; j<4; j++) {

printf("%x\n", \*(p+j));

}

输出为：

Low address--->High address

10

41

12

ffffff81

## 左移先移lsb， 右移先移msb。 与endian无关

int i = 0x81124110;

printf("Low byte---->High byte\n");

for (int j=0; j<4; j++) {

printf("%d: byte is 0x%x\n", j, (i>>j\*8) & 0xff);

}

输出为：

Low byte---->High byte

0: byte is 0x10

1: byte is 0x41

2: byte is 0x12

3: byte is 0x81

## Ntohl和htonl

**这2个函数在”主机序和网路序一样的机器上“， do nothing。**

**这2个函数在”主机序和网络序不一样的机器上“， 将字节反转。**

//2016.5.11

char ch = 0x1c;

int i\_ch = (int)ch;

printf("i\_ch is %x\n", i\_ch);

printf("htonl(i\_ch) is %x\n", htonl(i\_ch)); //reverse here(in Intel), do nothing in PowerPC since network order is the same as host order in PowerPC

printf("ntohl(i\_ch) is %x\n", ntohl(i\_ch)); //same as above

int n\_h\_ich = ntohl(htonl(i\_ch));

int h\_n\_ich = htonl(ntohl(i\_ch));

int n\_n\_ich = ntohl(ntohl(i\_ch));

int h\_h\_ich = htonl(htonl(i\_ch));

printf("n\_h\_ich is %x, h\_n\_ich is %x, n\_n\_ich is %x, h\_h\_ich is %x\n", n\_h\_ich, h\_n\_ich, n\_n\_ich, h\_h\_ich);

}

打印如下：

i\_ch is 1c

htonl(i\_ch) is 1c000000

ntohl(i\_ch) is 1c000000

n\_h\_ich is 1c, h\_n\_ich is 1c, n\_n\_ich is 1c, h\_h\_ich is 1c

## 在函数内部定义新函数方法

int Socket::writen(const void \*buf, unsigned int len, int timeout) {

struct pollfd pfd; // structure to poll()

unsigned int wrtd = 0; // total data written

unsigned int wnow = 0; // partial data written

ssize\_t ret = len;

// set poll events

pfd.fd = mSocket;

pfd.events = POLLOUT;

pfd.revents = 0;

LogToFile::writeIntln("Ater-receive-sp-", "", len);

LogToFile::writeStringln("Ater-receive-sp-", "", "writen1");

struct sigaction sa;

struct kludge {

static void my\_sighandler(int sig) {

LogToFile::writeStringln("Ater-receive-sp-", "", "SIGPIPE happened");

}

};

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");

}

## C++中强行将int转换为char结果为“截取低位， 舍弃高位”。 此处高位指most significant byte.

## C++ 将char 转换为int结果是一样的， 只是高位添0， 此处高位置msb。

http://stackoverflow.com/questions/5585532/c-int-to-byte-array ： convert int to byte array.

## 栈空间是向下增长的， 堆空间是向上增长的

## 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

**Struct内的struct的初始化， 及struct里的vector的初始化。这样的extend初始化值只能在C++11中进行。**

#include <vector>

#include <string>

#include <cstdio>

struct ST1 {

double a;

double b;

std::vector<std::string> vec;

};

struct ST2 {

int i;

ST1 st1;

};

int main() {

ST1 st1 = {0.1, 2, {"fun", "foo"} };

printf("st1.vec[1] is %s\n", st1.vec[1].c\_str());

ST2 st2 = {1, {0.1, 2, {"fun", "hello"}} };

printf("st2.st1.vec[1] is %s\n", st2.st1.vec[1].c\_str());

return 0;

}

~

## 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 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 { ... };

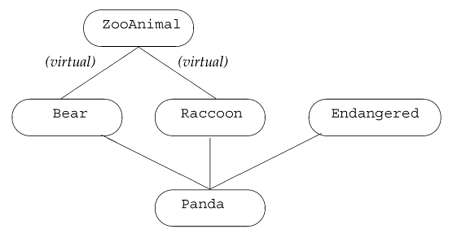
class ostream : virtual public ios { ... };

// 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， 却要负责复制/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*

## 引用作为构造函数的参数和对象作为构造函数的参数的区别

//To summary,

//Take object:Bar as copy-constructor parameter, then copy-constructing of bar will be called.

//Take reference:Bar as copy-constructor parameter, then copy-constructing of bar won't be called.

//if Foo::bar is object, then bar(b) will call Bar's copy-constructor.

//if Foo::bar is reference, then bar(b) won't call Bar's copy-constructor.

构造函数入参：引用。成员：引用。

入参的copy-constructor一次都不会被调用

get这个成员的那个函数返回的也最好是引用。

这个目的是为了成员随外界变化而变化。

构造函数入参：非引用。 成员：非引用。

入参时，调用copy-constructor

成员初始化时， 调用copy-constructor

成员不随外界变化而变化。

构造函数入参：引用， 成员为非引用。（不推荐）

入参时，入参的拷贝构造函数不被调用。

成员初始化时。入参的copy-constructor会被调用。

成员不会随外界变化而变化。

构造函数入参：非引用， 成员为引用。（更不推荐）

入参时，入参的拷贝构造函数将被执行一遍。

成员初始化时，入参的copy-constructor不会被调用。成员是入参时临时生成的object的别名。

成员不会随外界变化而变化。

-bash-4.1$ cat constructorWithReferenceParameter.cpp

#include <cstdio>

class Bar {

public:

Bar() {

printf("constructing Bar\n");

}

Bar(Bar& src) {

printf("copy-constructor of Bar\n");

}

int getNum() {

return num;

}

int setNum(int n) {

num = n;

return num;

}

private:

int num;

};

class Foo {

public:

//To summary,

//Take object:Bar as copy-constructor parameter, then copy-constructing of bar will be called.

//Take reference:Bar as copy-constructor parameter, then copy-constructing of bar won't be called.

//if Foo::bar is object, then bar(b) will call Bar's copy-constructor.

//if Foo::bar is reference, then bar(b) won't call Bar's copy-constructor.

Foo(Bar b): bar(b), r\_bar(b) { printf("constructing foo(Bar)\n"); } //这种用法不好， 因为r\_bar是Foo构造函数里面的临时生成的对象的别名。

Foo(int i, Bar& b): bar(b), r\_bar(b) { printf("constructing foo(int, Bar&)\n"); }

void fun(Bar& b) { printf("fun(Bar& b)\n"); }

void havefun(Bar b) { printf("havefun(Bar b)\n"); }

Bar& getReferenceBar() {

return r\_bar;

}

private:

Bar bar;

Bar& r\_bar;

};

int main() {

Bar b;

b.setNum(11);

printf("b.num is %d\n", b.getNum());

printf("---------------------------\n");

Foo f(b);

printf("---------------------------\n");

Foo f1(1, b);

printf("f1::b::num is %d\n", f1.getReferenceBar().getNum());

printf("---------------------------\n");

f.fun(b);

printf("---------------------------\n");

f.havefun(b);

printf("---------------------------\n");

b.setNum(12);

printf("f1::b::num is %d after change\n", f1.getReferenceBar().getNum());

return 0;

}

-bash-4.1$

## 临时object都是右值，不能bond至non-const 变量

temporary objects are rvalue, and can't be bound to reference to non-const

class Hei {

public:

Hei() { printf("constructor of Hei\n"); }

Hei(Hei& h) {

printf("copy-constructor of Hei\n");

}

Hei& operator=(const Hei& src) {

printf("operator= of Hei\n");

return \*this;

}

Hei getHei() {

return \*this;

}

};

代码：

Hei hei;

Hei h = hei; //此句是OK的

Hei h1 = hei.getHei(); //此句不OK。因为getHei返回的临时object是rvalue， 其不能作为Hei(Hei& h)的参数。

修改： 将Hei(Hei& h)改为Hei(const Hei& h).

## 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*

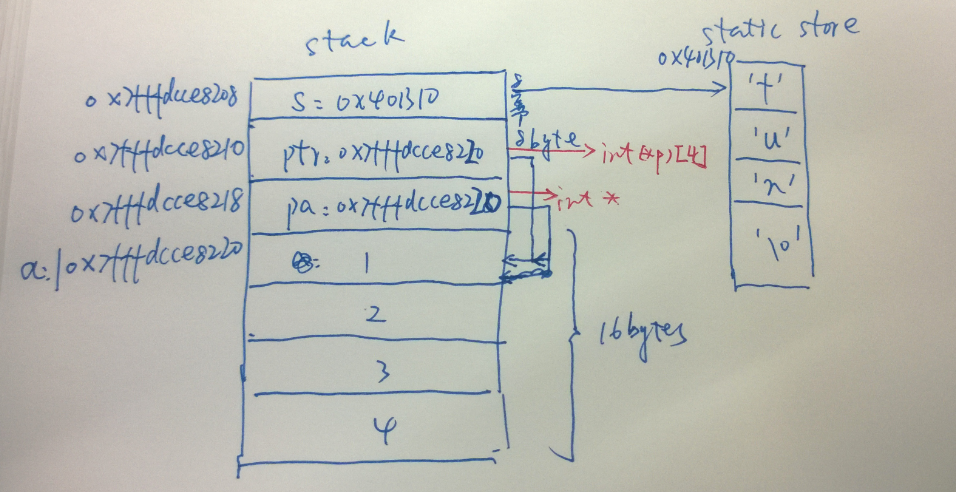
*s address is 0x40130f, s is fun, &s is 0x7fffdcce8208, &s[1] is 0x401310, \*s is f*

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， 宜使用引用传递， 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。Const object只能调用const 成员函数。 但非const object可以调用const 成员函数。const与否可以决定是否重载。**

#include <stdio.h>

class Bar {

public:

void constF() const {

printf("I am in Bar::constF()\n");

const Foo const\_foo = Foo();

}

};

class Hey {

public:

void F() {

printf("I am in Hey::F()\n");

}

};

class Foo {

public:

void F() {

printf("I am in Foo::F()\n");

}

**//C++ standard allows overloading of functions that differ only by the constness no matter const** qualify function parameter or

return type

void F() const {

printf("I am in Foo::F() const\n");

}

void fun(const Foo &object) {

printf("I am in fun(const Foo&)\n");

}

void fun(Foo &object) {

printf("I am in fun(Foo &)\n");

}

};

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

Bar bar;

bar.constF(); //non-const object can call const function.

const Hey hey = Hey();

//hey.F(); //Wrong, const object can not call non-const function.

Foo nonConst\_foo;

const Foo const\_foo = Foo();

nonConst\_foo.F(); //non-const object select non-const function firstly

const\_foo.F(); //const object select const function firstly

nonConst\_foo.fun(nonConst\_foo);

nonConst\_foo.fun(const\_foo); //non-const object can call const-function also.

输出为：

I am in Bar::constF()

I am in Foo::F()

I am in Foo::F() const

I am in fun(Foo &)

I am in fun(const Foo&)

## 成员变量在Const成员函数里转变为const类型了， 如果返回类型是引用或者指针，最好也以const修饰。

-bash-4.1$ cat canConstMemberFunctionReturnNonConstType.cpp

#include <stdio.h>

#include <string>

//In a const-function, every data member becomes const in such way that it cannot be modified.

class Person {

public:

Person(std::string name, int age, int score): name(name), age(age), score(score) { salary = new int(0); }

const std::string& name\_good() const { // Right: the caller can't change the Person's name

return name;

}

//Wrong, the caller can change the Person's name.

//Compile point of view, it is NOK for std::string& X = const std::string name.

/\*

std::string& name\_evil() const {

return name;

}

\*/

//Right. The caller can not change the Person's age.

//Compile point of view, it is OK for int x = const int age.

int get\_age() const {

return age;

}

//Wrong, the caller can change the Person's score.

//Compile point of view, NOK for int &x = const int age;

/\*

int & get\_score() const {

return score;

}

\*/

//seems wrong, since the caller can change the Person's salary. But it is right since const here means caller can not change the value of returned pointer, but not the value to which the pointer points though it is not what we expected.

//Compile OK. since "int \*salary" become "int \* const salary" but not "const int \*salary". It is OK for int \*x = int \* const salary

int\* get\_salary() const {

return salary;

}

~Person() {

delete salary;

}

private:

std::string name;

int age;

int score;

int \*salary;

};

void myCode(const Person& p) // myCode() promises not to change the Person object...

{

// p.name\_evil() = "Igor"; // But myCode() changed it anyway!!

// printf("p.name\_evil() is %s\n", p.name\_evil().c\_str());

int \*pointer = p.get\_salary();

printf("p.get\_salary() is %d\n", \*(p.get\_salary()));

\*pointer = 30000;

printf("p.get\_salary() is %d\n", \*(p.get\_salary()));

pointer = new int(12); //ok here, since asigning pointer by another pointer is value-transmit

printf("p.get\_salary() is %d\n", \*(p.get\_salary()));

}

int main() {

Person p("xiaoming", 11, 99);

myCode(p);

}

输出为：

-bash-4.1$ ./canConstMemberFunctionReturnNonConstType.o

p.get\_salary() is 0

p.get\_salary() is 30000

p.get\_salary() is 30000

## POD class type //似乎就是个概念，除了说non-POD需要用户自定义的构造函数。

POD: plain old data。

### Very informally:

A POD is a type (including classes) where the C++ compiler guarantees that there will be no "magic" going on in the structure: for example hidden pointers to vtables, offsets that get applied to the address when it is cast to other types (at least if the target's POD too), constructors, or destructors. Roughly speaking, a type is a POD when the only things in it are built-in types and combinations of them. The result is something that "acts like" a C type.

### Less informally:

* int, char, wchar\_t, bool, float, double are PODs, as are long/short and signed/unsigned versions of them.
* pointers (including pointer-to-function and pointer-to-member) are PODs,
* enums are PODs
* a const or volatile POD is a POD.
* a class, struct or union of PODs is a POD provided that all non-static data members are public, and it has no base class and no constructors, destructors, or virtual methods. Static members don't stop something being a POD under this rule. //类里所有non-static数据成员均为public哦，且没有自定义的构造函数， 析构函数，虚函数， 且没有基类。
* Wikipedia is wrong to say that a POD cannot have members of type pointer-to-member. Or rather, it's correct for the C++98 wording, but TC1 made explicit that pointers-to-member are POD.

## Foo \*\* 不能对const Foo\*\*赋值， 但能对const Foo \* const \*赋值

-bash-4.1$ cat const\_cast\_\_pointerToPointer.cpp

#include <stdio.h>

class Foo {

public:

void pseudo\_modify() const

{

printf("just print, no modify\n");

}

};

void f(const Foo \*\*p) {

printf("I am f(const Foo \*\*p)\n");

}

void g(const Foo \* const \*p) {

printf("I am g(const Foo \* const \*p)\n");

}

int main()

{

Foo F;

Foo \*p\_F = &F;

Foo \*\*pp\_test = &p\_F;

// f(pp\_test); //Error: it is ilegal and immoral to convert Foo\*\* to const Foo\*\*

g(pp\_test); //Okay: it is legal and moral to convert Foo\*\* to const Foo\* const \*

//Reason explained.

const Foo X = Foo();

Foo Y;

const Foo \*p = &Y;

//const Foo \*pFoo; //correct modify.

Foo \*pFoo; //Wrong here.

const Foo \*\*pp = &pFoo; //Seems this OK since it is OK to assign const by non-const

\*pp = &X; //Right. \*pp is the value of pp, which is pFoo here.

(\*\*pp).pseudo\_modify(); //obvious Wrong since \*\*pp is "const Foo", can not modify const.

(\*pp)->pseudo\_modify(); //seems OK, \*pp is pFoo, pFoo is non-const, so can call modify.

//summary: Why

//const Foo \*const\_p = &Y;---- OK.

//But const Foo \*\*pp = &pFoo---NOK.

//The reason is that const Foo \*\*pp means \*\*pp is const Foo, so can not use (\*\*pp).modify(),

//But does not gurantee \*pp points to const Foo, so can call \*pp->modify() theretically.

}

## 如果某个类没有用户定义的default constructor， 定义此类const 类型的object时候，需要显示初始化这个object。

**if the object is of const-qualified type, the underlying class type shall have a user-declared default constructor. Otherwise,** if no initializer is specified for a nonstatic object, the object and its subobjects, if any, have an indeterminate initial value; **if the object or any of its subobjects are of const-qualified type, the program is ill-formed.**

**class Foo {**

**};**

class Bar {

public:

Bar() {}

};

int main() {

Foo foo\_object;

//const Foo const\_foo\_object; //wrong, need explicte initialise

const Foo const\_foo\_object = Foo();

Bar bar\_object;

const Bar const\_bar\_object; //ok, since Bar has its own user-define default constrcutor

}

## 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\_pton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;

};

struct in\_addr {

uint32\_t s\_addr; /\* address in network byte order \*/

};

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

struct in6\_addr {

unsigned char s6\_addr[16]; /\* IPv6 address \*/

};

### 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);

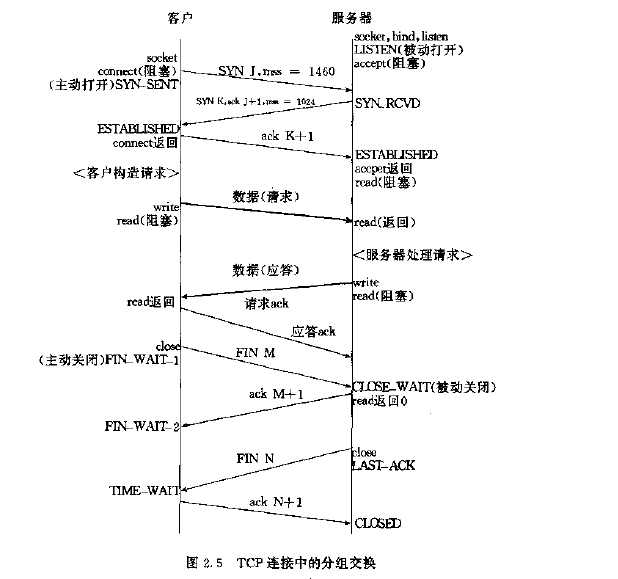
}

}

### 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



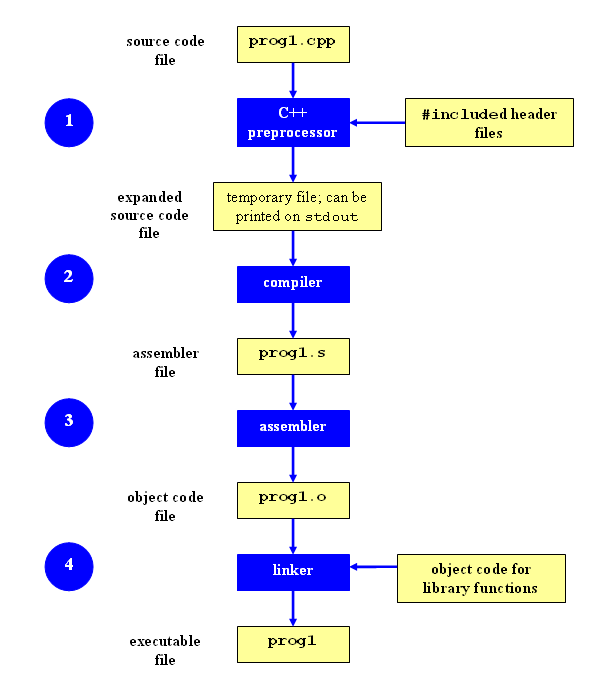
### 一个ip:port允许拥有多个socket, 例如某个网站使用以80端口接收。到底允许有多少个socket取决于listen的backlog值大小。

## 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。

## 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中读pcap文件

File file = new File(path);

InputStream is = new FileInputStream(file);

byte[] buffer\_4 = new byte[4];

byte[] bufferFileHdr = new byte[PCAP\_FILE\_HEADER\_LEN];

byte[] bufferPktHdr = new byte[PCAP\_PKT\_HEADER\_LEN];

// unpack

try

{

// read pcap file header

int m = is.read(bufferFileHdr); //InputStream.read(byte[] b), 从inputstream读取内容， 内容长度为b.length。或知道文件结尾。等同与InputStream.read(b, 0, b.length);

if (m != PCAP\_FILE\_HEADER\_LEN)

throw new Exception("read pcap file header error");

while (true) {

// read packet header

m = is.read(bufferPktHdr);

checkReadLen(m, PCAP\_PKT\_HEADER\_LEN);

// read packet len

m = is.read(buffer\_4);

checkReadLen(m, 4);

reverseByteArray(buffer\_4);

int pktlen = byteArrayToInt(buffer\_4, 0);

byte[] content = new byte[pktlen];

m = is.read(content);

checkReadLen(m, pktlen);

## 创建final map， final list

private static final Map<String, Integer> paddingmap;

static {

Map<String, Integer> aMap = new HashMap<String, Integer>();

aMap.put("Pro1", 16);

aMap.put("Pro2", 16);

aMap.put("Pro3", 32);

aMap.put("Pro5", 16);

paddingmap = Collections.unmodifiableMap(aMap);

}

private static final List<String> proNames;

static {

proNames = Collections.unmodifiableList(Arrays.asList("pro1", "pro2", "pro3", "pro5"));

}

## 补充字符串：

public static String repeatString(String s, int count) {

return new String(new char[count]).replace("\0", s);

}

## 如何在mainthread外起一个新进程去运行另外的工作

Create a class that implements the Runnable interface. Put the code you want to run in the run() method - that's the method that you must write to comply to the Runnable interface. In your "main" thread, create a new Thread class, passing the constructor an instance of your Runnable/传入一个implement了Runnable interface的class的实例, then call start() on it. start tells the JVM to do the magic to create a new thread, and then call your run method in that new thread.

public class MyRunnable implements Runnable {

private int var;

public MyRunnable(int var) {

this.var = var;

}

public void run() {

// code in the other thread, can reference "var" variable

}

}

public class MainThreadClass {

public static void main(String args[]) {

MyRunnable myRunnable = new MyRunnable(10);

Thread t = new Thread(myRunnable) //Thread有构造函数Thread(Runnable target).

t.start(); //调用start()会调用run函数.run函数如下，所以target的run函数被调用

}

}

@Override

public void run() {

if (target != null) {

target.run();

}

}

另一个方法：

public class HelloThread extends Thread {

public void run() { //Thread也implement了Runnable interface， 原因不明

System.out.println("Hello from a thread!");

}

public static void main(String args[]) {

(new HelloThread()).start(); //Thread还有个构造函数为Thread().然后调用start函数。start函数调用run函数()，注意此时的run函数被重写了即为HelloThread里的run。

}

}

## Double brace initialize

List<MyClass> list = new ArrayList<MyClass>() {

{

add(new MyClass("name", "address", 23));

add(new MyClass("name2", "address2", 45));

}

};

定义class同时实例化一个对象：

Thread t = new Thread() {

public void run() {

System.out.println("blah");

}

};

t.start();

## 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

## JAVA函数调用时：primitive采用值传递， object采用类C的指针传递

注意String也是Object，但由于String类没有改变自身的方法，所以不能通过类C指针传递来改变值。

代码解释见/home/hongjyan/java/dog下的文件

Let me explain this through an [example](http://stackoverflow.com/a/9404727/597657):

public class Main{

public static void main(String[] args){

Foo f = new Foo("f");

changeReference(f); // It won't change the reference!

modifyReference(f); // It will modify the object that the reference variable "f" refers to!

}

public static void changeReference(Foo a){

Foo b = new Foo("b");

a = b;

}

public static void modifyReference(Foo c){

c.setAttribute("c");

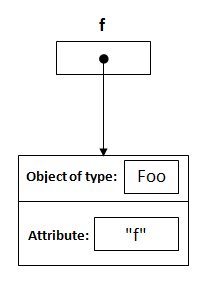
}

}

I will explain this in steps:

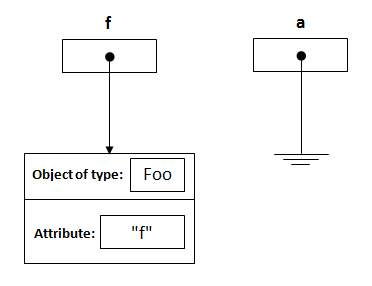
1. Declaring a reference named f of type Foo and assign it to a new object of type Foo with an attribute "f".

Foo f = new Foo("f");



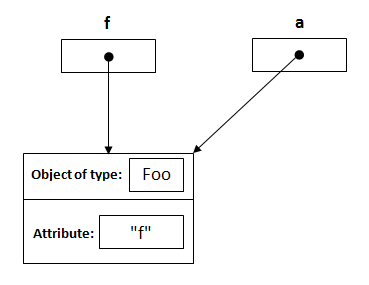
1. From the method side, a reference of type Foo with a name a is declared and it's initially assigned to null.

public static void changeReference(Foo a)



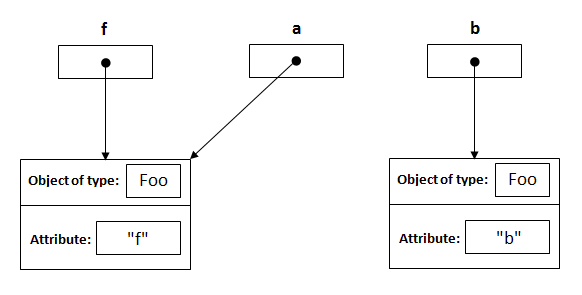
1. As you call the method changeReference, the reference a will be assigned to the object which is passed as an argument.

changeReference(f);

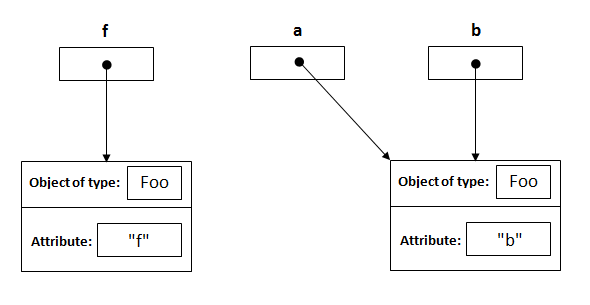


1. Declaring a reference named b of type Foo and assign it to a new object of type Foo with an attribute "b".

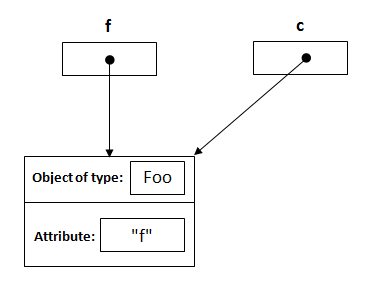
Foo b = new Foo("b");



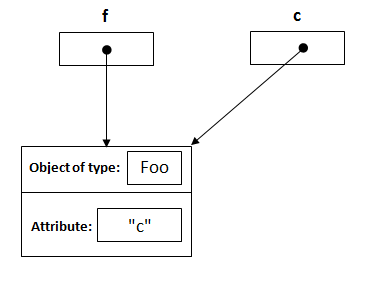
1. a = b is re-assigning the reference a NOT f to the object whose its attribute is "b".



1. As you call modifyReference(Foo c) method, a reference c is created and assigned to the object with attribute "f".



1. c.setAttribute("c"); will change the attribute of the object that reference c points to it, and it's same object that reference f points to it.



I hope you understand now how passing objects as arguments works in Java :)

## JAVA中可以返回函数内部创建的数组

public class VInF {

public static String[] F() {

String[] str = new String[2];

str[0] = "Hello";

str[1] = "World";

return str;

}

public static void main(String[] args) {

String[] strout = F();

System.out.println(str[0] + " " + str[1]);

}

}

String[] str的内存不会被释放，String[] strout = F()是将reference to str的值拷贝给了strout。如果想直接拷贝值的话， 可以用String[] strcpy = System.arraycopy(F()).

C++这样的做法是错误， 因为函数内部的local variable的内存， 出了函数之后会被释放掉。

# Make file

## 编译过程

无论C, C++， 首先把源文件编译成中间代码文件， 在windows下为.obj文件，在unix下为.o文件， 及object File, 这个动作叫做编译， 这个过程中，编译器需要源文件语法正确，函数和变量的声明正确，对于后者，通常只需告诉编译器头文件的所在位置(头文件中应该只是声明， 而地难以应该放在C/C++文件中)，如果函数未声明，编译器会给一个警告但可以生成object File，但是在链接时会在所有的object File中找寻函数的实现，如果找不到，那么会报链接错误码（Link 2001）。 然后将大量的object file合成执行文件，这个动作叫做链接（link）. 链接时， 主要是链接函数和全局变量， 所以可以使用这些中间目标文件来链接我们的应用程序。 链接器不管函数所在的源文件， 只管函数的中间目标文件，在大多数的时候，由于源文件太多， 编译生成的中间目标文件太多， 而在链接是需要明显地指明中间目标文件名， 这对于编译很不方便， 所以，我们要给中间目标文件打个包，在Windows下这种包叫做库文件，也就是lib文件， 在unix下，是archive文件，也就是.a文件。

## Makefile的目的

1. 如果这个工程没有编译过，那么我们所有的C文件都要编译并被链接。
2. 如果这个工程的某几个C文件被修改，那么我们只编译被修改的C文件，并链接目标程序。
3. 如果这个工程的头文件被改变了，那么我们需要编译引用了这几个头文件的C文件，并链接目标程序。

## Makefile的原则

Target: prerequisites

Command

Target通常是一个目标文件，可以是object File，也可以是执行文件。还可以是一个标签。

Prereqiusites是要生成那个target所需要的文件或者目标。

Command也就是make需要执行的命令。

如果prerequisites比target要新， 或者target不存在的话，那么command 被执行， command一般使用UNIX的标准Shell， 也就是/bin/sh来执行命令。

## Make是如何工作的

edit : main.o kbd.o command.o display.o  insert.o search.o files.o utils.o

            cc -o edit main.o kbd.o command.o display.o   insert.o search.o files.o utils.o

  main.o : main.c defs.h

 cc -c main.c

kbd.o : kbd.c defs.h command.h

cc -c kbd.c

command.o : command.c defs.h command.h

cc -c command.c

display.o : display.c defs.h buffer.h

cc -c display.c

insert.o : insert.c defs.h buffer.h

cc -c insert.c

search.o : search.c defs.h buffer.h

cc -c search.c

files.o : files.c defs.h buffer.h command.h

 cc -c files.c

utils.o : utils.c defs.h

cc -c utils.c

clean :

rm edit main.o kbd.o command.o display.o  insert.o search.o files.o utils.o

在默认情况下，我们只输入make命令。那么

1. Make会在当前目录下找名字叫”Makefile”或“makefile”的文件
2. 如果找到，它会找文件中的第一个目标文件，在上面的例子中，他会找到edit这个文件，并把这个文件作为最终目标文件。
3. 如果edit文件不存在，或者edit所依赖的后面的.o文件的文件修改时间要比edit这个文件新，那么，他会执行后面的所定义的命令来生成edit这个文件。
4. 如果edit所依赖的.o文件也不存在，那么make会在当前文件中找目标为.o文件的依赖性，如果找到则再根据那一个规则生成.o文件，这有点像一个堆栈的过程。
5. 当然，.c和.h文件是存在的，于是make会生成.o文件，然后用.o文件声明make的终极任务，也就是执行文件edit了。

这就是整个make的依赖性， make会一层又一层地去寻找文件的依赖关系，知道最终编译出第一个目标文件。 在找寻的过程中，如果出现错误，比如最后被依赖的文件找不到，那么make会直接退出，并报错，而对于所定义的命令的错误，或者编译不成功，make根本不理。

## Makefile里面有什么

1. 显示规则。显示规则说明了，如何生成target文件
2. 隐晦规则： 由于make有自动推导的功能，所以有隐晦规则
3. 变量定义： 为了更方便地书写makefile
4. 文件指示： 其包含了3个部分， 一个是在一个makefile中引用另一个makefile， 就想c语言中的include一样， 如include foo.make \*.mk；另一个是指根据某些情况制定makefile中的有效部分，就想C语言中的预编译#if一样；还有就死定义一个多行的命令
5. 注释: 以#开头， 注意makefile中的命令，必须以Tab建开始

## Makefile的文件名

make命令，会在当前目录下按顺序寻找文件名为GNUmakefile、makefile、Makefile文件，推荐使用Makefile这个文件名。 当然也可以使用其他的文件名, 如Make.Linux、Make.Solaris、Make.AIX等，如果要指定特定的Makefile，可以私用make的-f、--file参数， 如：make –f Make.Linux或者make –file Make.AIX

## Make如何引用的其他文件

Make会在下面的目录中寻找文件， 如果文件没找到的话， make会生成一条警告信息，但不会马上出现致命错误，他会继续载入其他的文件，一旦完成makefile的读取，make才会出现一条致命信息， 如果不想让make理那些无法读取的文件，而继续执行，可以在include前加一个减号”-“。如-include <filename>.

### 现在当前目录下首先寻找。

### 如果有-I、--include-dir参数， 那么make就会在这个参数所指定的目录下去寻找。

### 如果目录/include存在的话， make也会去找。

### 环境变量MAKEFILE—不推荐使用

如果当前环境中定义了环境变量MAKEFILES, 那么， make会把这个变量中的值做为一个类似于include的动作。注意这个变量中的值是其他的Makefile， 用空格分开， 只是它和include不同的是， 从这个环境变量中引入的Makefile的目标不会起作用，如果环境变量中定义的文件发现错误， make也会不理。

不推荐使用这个变量，因为定义了这个变量后， 所有的Makefile都会受到它的影响。

### VPATH环境变量

Make在寻找文件时，如果在当前的文件中找不到的话，如果定义了VPATH这个变量， 它会到VPATH这个变量定义的值里面去查找的。如VPATH = src:../headers， 上面定义指定了2个目录， “src”和“../headers”, make 会按照这个顺序进行搜索。目录由冒号分隔， 当然，当前目录永远是最高优先搜素的地方。

### vpath变量

这个比VPATH灵活， 它可以指定不同的文件在不同的搜索目录中。它的使用方法有3种。

#### vpath <pattern> <directories>

为符合模式<pattern>的文件指定搜索目录<directories>

如vpath %.h ../headers

%表明匹配零个或若干字符。 %.h表示所有以”.h”结尾的文件。表明在../headers目录下寻找 “.h”结尾的文件。

vpath %.c foo:bar 表明在“.c”结尾的文件，现在”foo”目录，然后在”bar”目录中查找。

#### vpath <pattern>

清除符合模式<pattern>的文件的搜索目录

#### vpath

清除所有已被设置好了的文件搜索目录

## Makefile中还可以使用变量

如上的makefile，可以定义如下:

Objects = main.o kbd.0 command.o display.o insert.o search.o files.o utils.o

下面我们就可以

Edit：$(objects)

## 隐含规则

GNU的make只要看到一个.o文件，它会自动将.c文件加到依赖关系中。例如如果make找到一个whatever.o, 那么whatever.c就会被自动作为whatever.o的依赖文件。

隐含规则会使用一些我们系统变量，我们可以通过改变这些系统变量的值来定制隐含规则的运行时的参数。如系统变量”CFLAGS”可以控制编译时的编译器参数。同样我们可以通过使用”模式规则”或“后缀规则”用来写下自己的隐含规则。

### 隐含规则的优先级

注意在make的隐含规则库中，每一条隐含规则都在库中尤其顺序， 越靠前的则是越被经常使用的，所以这会导致有时我们显示指定了目标，依赖目标也不会管。如下:

Foo.o: foo.p

这个只定义了目标依赖，但是没有定义命令行。 如果目录下存在了foo.c 文件， 由于隐含规则foo.o : foo.c 比foo.o: foo.p优先级要高， 所以这用户显示指定的依赖关系始终不会得到执行。 要消除这种情况， 必须在foo.o:foo.p 下面写上命令行。

### 隐含规则一览

我们可以使用make中的-r或者—no-builtin-rules选项来取消所有的预设置的隐含规则。

但是即使我们指定了-r参数， 某些隐含规则还会生效，因为有许多的隐含规则都使用了”后缀规则”来定义的， 所以， 只要隐含规则中有”后缀列表“（也就一系统定义在目标.SUFFIXES的依赖目标）， 那么隐含规则就会生效。 默认的后缀列表是：.out, .a, .ln, .o, .c, .cc, .C, .p, .f, .F, .r, .y, .l, .s, .S, .mod, .sym, .def, .h, .info, .dvi, .tex, .texinfo, .texi, .txinfo, .w, .ch .web, .sh, .elc, .el。

1、编译C程序的隐含规则。   
“<n>;.o”的目标的依赖目标会自动推导为“<n>;.c”，并且其生成命令是“$(CC) –c $(CPPFLAGS) $(CFLAGS)”   
  
2、编译C++程序的隐含规则。   
“<n>;.o”的目标的依赖目标会自动推导为“<n>;.cc”或是“<n>;.C”，并且其生成命令是“$(CXX) –c $(CPPFLAGS) $(CFLAGS)”。（建议使用“.cc”作为C++源文件的后缀，而不是“.C”）   
  
3、编译Pascal程序的隐含规则。   
“<n>;.o”的目标的依赖目标会自动推导为“<n>;.p”，并且其生成命令是“$(PC) –c  $(PFLAGS)”。   
  
4、编译Fortran/Ratfor程序的隐含规则。   
“<n>;.o”的目标的依赖目标会自动推导为“<n>;.r”或“<n>;.F”或“<n>;.f”，并且其生成命令是:   
    “.f”  “$(FC) –c  $(FFLAGS)”   
    “.F”  “$(FC) –c  $(FFLAGS) $(CPPFLAGS)”   
    “.f”  “$(FC) –c  $(FFLAGS) $(RFLAGS)”

5、预处理Fortran/Ratfor程序的隐含规则。   
“<n>;.f”的目标的依赖目标会自动推导为“<n>;.r”或“<n>;.F”。这个规则只是转换Ratfor或有预处理的Fortran程序到一个标准的Fortran程序。其使用的命令是：   
    “.F”  “$(FC) –F $(CPPFLAGS) $(FFLAGS)”   
    “.r”  “$(FC) –F $(FFLAGS) $(RFLAGS)”   
  
6、编译Modula-2程序的隐含规则。   
“<n>;.sym”的目标的依赖目标会自动推导为“<n>;.def”，并且其生成命令是：“$(M2C) $(M2FLAGS) $(DEFFLAGS)”。“<n.o>;” 的目标的依赖目标会自动推导为“<n>;.mod”，并且其生成命令是：“$(M2C) $(M2FLAGS) $(MODFLAGS)”。   
  
7、汇编和汇编预处理的隐含规则。   
“<n>;.o” 的目标的依赖目标会自动推导为“<n>;.s”，默认使用编译品“as”，并且其生成命令是：“$(AS) $(ASFLAGS)”。“<n>;.s” 的目标的依赖目标会自动推导为“<n>;.S”，默认使用C预编译器“cpp”，并且其生成命令是：“$(AS) $(ASFLAGS)”。   
  
8、链接Object文件的隐含规则。   
“<n>;”目标依赖于“<n>;.o”，通过运行C的编译器来运行链接程序生成（一般是“ld”），其生成命令是：“$(CC) $(LDFLAGS) <n>;.o $(LOADLIBES) $(LDLIBS)”。这个规则对于只有一个源文件的工程有效，同时也对多个Object文件（由不同的源文件生成）的也有效。例如如下规则：   
  
    x : y.o z.o   
  
并且“x.c”、“y.c”和“z.c”都存在时，隐含规则将执行如下命令：   
  
    cc -c x.c -o x.o   
    cc -c y.c -o y.o   
    cc -c z.c -o z.o   
    cc x.o y.o z.o -o x   
    rm -f x.o   
    rm -f y.o   
    rm -f z.o   
  
如果没有一个源文件（如上例中的x.c）和你的目标名字（如上例中的x）相关联，那么，你最好写出自己的生成规则，不然，隐含规则会报错的。   
  
9、Yacc C程序时的隐含规则。   
“<n>;.c”的依赖文件被自动推导为“n.y”（Yacc生成的文件），其生成命令是：“$(YACC) $(YFALGS)”。（“Yacc”是一个语法分析器，关于其细节请查看相关资料）   
  
10、Lex C程序时的隐含规则。   
“<n>;.c”的依赖文件被自动推导为“n.l”（Lex生成的文件），其生成命令是：“$(LEX) $(LFALGS)”。（关于“Lex”的细节请查看相关资料）   
  
11、Lex Ratfor程序时的隐含规则。   
“<n>;.r”的依赖文件被自动推导为“n.l”（Lex生成的文件），其生成命令是：“$(LEX) $(LFALGS)”。   
  
12、从C程序、Yacc文件或Lex文件创建Lint库的隐含规则。   
“<n>;.ln” （lint生成的文件）的依赖文件被自动推导为“n.c”，其生成命令是：“$(LINT) $(LINTFALGS) $(CPPFLAGS) -i”。对于“<n>;.y”和“<n>;.l”也是同样的规则。

## 隐含规则使用的变量

在隐含规则中的命令中， 基本上都是使用了一些预先设置的变量。 可以在makefile或者make的命令行里面改变这些变量值， 这样在使用隐含规则进行编译时，就会起作用了。

例如编译C程序的隐含规则的命令是”$(CC) –c $(CFLAGS) $(CPPFLAGS)”。 Make中默认的编译命令是cc，如果将变量$(CC)重定义为”gcc”, 把变量”$(CFLAGS)”重定义为”-g”，那么隐含规则的命令就变为”gcc –c –g $(CPPFLAGS)”的样子执行了。

我们可以将隐含规则中的变量分成2种： 一种是命令相关的，如”CC”; 一种是参数相关的，如“CFLAGS”。

### 关于命令的变量

AR: 函数库打包程序。默认命令是”ar”

AS : 汇编语言编译程序。默认命令是”as”.

CC: c语言编译程序。默认命令是“cc”

CXX: C++语言编译程序，默认命令是“g++“

CO: 从RCS文件中扩展文件程序。默认命令是“co“

CPP: C程序的预处理器（输出是标准输出设备）。默认命令是”$(CC) –E”.

FC: Fortran 和Ratfor的编译器的预处理程序。默认命令是“f77“

GET: 从SCCS文件中扩展文件的程序。默认命令是”get”.

LEX: Lex方法分析器程序（针对C或者Ratfor）。默认命令是lex。

PC: Pascal语言编译程序，默认命令是“pc“

YACC: Yacc文法分析器（针对Ratfor程序）。默认命令是“yacc –r”.

MAKEINFO: 转换Textinfo源文件（.texi）到info文件程序。默认命令是“makeinfo”。

TEX: 从Tex源文件创建Tex DVI文件的程序。默认命令是“tex“

TEXI2DVI ：从Textinfo源文件创建Tex DVI文件的程序。默认命令是“texi2dvi“。

WEAVE: 转换Web到Tex的程序。默认命令是“weave“。

CWEAVE: 转换C Web到Tex的程序。默认命令是“cweave“。

TANGLE: 转换Web到Pascal语言的程序。 默认命令是”tangle”.

CTANGLE: 转换C Web到C。默认命令是“ctangle“。

RM: 删除文件命令。默认命令是“rm –f”.

### 关于命令参数的变量

下面的这些变量都是相关上面的命令的参数。如果没有指明其默认值， 那么其默认值都是空。

ARFLAGS: 函数库打包程序AR命令的参数。默认值是“rv”。

ASFLAGS: 汇编语言编译器参数。

CFLAGS: C语言编译器参宿

CXXFLAGS: C++语言编译器参数。

COFLAGS: RCS命令参数。

CPPFLAGS: C预处理器参数。（C和Fortan编译器也会用到）。

FFLAGS: Fortran语言编译器参数

GFLAGS: SCCS “get” 程序参数。

LDFLAGS: Lex文法分析器参数。

PFLAGS: Pascal语言编译器参数。

RFLAGS: Ratfor程序的Fortran编译器参数。

YFLAGS: Yaxx文法分析器参数。

## 取消隐含规则

%.o: %.s

只要写好规则，但是不于其下写上任何命令行即可。

## 隐含规则链

有时， 一个目标可能被一系列的隐含规则所用， 如， 一个.o 文件生成，可能先被Yacc的.y文件先 生成.c文件，然后在被C编译器生成。 我们把则一系列的隐含规则叫做”隐含规则链“，

在默认情况下中间目标和一般目标有2个地方不同： 第一个不同是除非中间的目标不存在，才会引发中间规则。 第二个不同的是： 只要目标成功产生， 那么产生最终目标过程中， 所产生的所有中间目标将被删除。可以阻止make自动删除中间目标。方法一：使用伪目标”.SECONDARY”来强制声明，如.SECONDARY : sec。 你还可以把你的目标，以模式的方式来指定，如%.o 成为伪目标”.PRECIOUS”的依赖目标。

在隐含规则链中，禁止同一个目标出现两次或者两次以上， 这样一来， 就可防止在make自动推导时出现无限递归的情况。

## 模式规则

s.%.o : %.k

Command

其表示以s.开头的、以..o结尾的文件需要同名的.k文件。

## 老式风格的”后缀规则”

### 基本定义

后缀规则是一个比较老式的定义隐含规则的方法。后缀规则会被模式规则逐步的取代。

双后缀规则定义了一对后缀:依赖文件的后缀和目标文件的后缀。如”.c .o”相当于”%.o : %.c”。

单后缀规则只定义了一个后缀，也就是源文件的后缀。如”.c”相当于”%: %.c”.

.c .o:

$(cc) –c $(CFLAGS) $(CPPFLAGS) –o $@ $<

等同于

.o:.c

$(cc) –c $(CFLAGS) $(CPPFLAGS) –o $@ $<

注意：

.c .o: foo.h

$(cc) –c $(CFLAGS) $(CPPFLAGS) –o $@ $<

这个是错的，如果使用后缀规则定义规则，那么就不要再使用模式规则了

### 后缀规则中，如果没有命令，那么就是毫无意义的， 因为它不会移去内建的隐含规则。

### make特定的后缀的添加和删除

我们可以使用伪目标”.SUFFIXES”来定义或是删除。

.SUFFIXES: .hack .win #将后缀.hack 和.win 加入后缀列表的末尾。

.SUFFIXES #删除默认的后缀

.SUFFIXES: .c .o .h #定义自己的后缀

Make的参数”-r“或”-no-builtin-rules”也会使默认的后缀列表为空。 而变量”SUFFIXE”被用来定义默认的后缀列表，可以同”.SUFFIXES”来改变后缀列表，但不要改变变量”SUFFIXE”的值。

## 多目标

如下kbd.o command.o files.o : command.h

## 静态模式规则

<targrts>： <target-pattern>: <prereq-pattern>

Targerts定义了一系列的目标文件。可以有通配符。是目标的一个集合。

Target-parrtern指明targets的模式， 也就是目标的一个集合。

Pre-patterns是指明目标的依赖模式，它对target-pattern形成的模式再进行一些依赖目标的定义。下例子：

Objects = foo.o bar.o

All : $(objects)

$(objects) : %.o : %.c

$(CC) –c $(CFLAGS) $< -o $@

上面的例子表示从object里面选取.o文件结尾的作为target, 选取同名.c文件结尾的作为pre-requist. 上例子等同于:

Foo.o : foo.c

$(CC) –c $(CFLAGS) foo.c –o foo.o

Bar.o : bar.c

$(CC) –c $(CFLAGS) bar.c –o bar.o

静态模式如果用得好了，很方便， 只用在object定义所有的内容。然后选用正确的静态模式即好了。

## 模式匹配

target ：prerequisite。 一般来说target和prerequisite里面都有%号， %号匹配一个或多个字符。 % 匹配出来的内容， 称为 “茎“， target中的”茎”会传递给prerequisite。 如:

%.o : %.c , 如果target匹配到了tar.o，那么茎就为tar， 那么prerequisite即为tar.c.

但是注意如果匹配出来的内容是是一个包含反斜杠的文件时，会先把目录给移除掉，然后传递茎， 然后再将目录给加回到prerequisite中。 如

c%r : e%t, 如果target匹配到了src/car了， 那么先移除src/, 茎即为a， 传递a到prerequisite，结合prerequisite里的模式，为eat， 加回目录，为src/eat.

## 自动化变量重要

如何书写一个命令来完成从不同的依赖文件生成相应的目标。因为在每一次的对模式规则的解析时， 都会是不同的目标和依赖文件。 自动化变量就是完成这个功能的。

### $@ 表示规则中的目标文件集。

在模式规则中，如果有多个目标，那么 $@就是匹配于目标中模式定义的集合。

### $% ----仅当目标是函数库文件中， 表示规则中的目标成员名。>>>目标成员名且该目标成员为函数库文件

如一个目标是” foo.a(bar.o)”, 那么”$%” 就是 “bar.o”, “$@”就是”foo.a”. 如果目标不是函数库文件(Unix 下是[.a], Windows下是[.lib]), 那么，其值为空。

### $< 依赖目标中的第一个目标名字。 如果依赖目标是以模式(即%)定义的， 那么”$<”将是符合模式的一系列的文件集。

注意， 其实一个一个取出来的。

### $?所有比目标新的依赖目标的集合。 以空格分隔。

### $^ 所有依赖目标的集合。 以空格分隔。 如果在依赖目标中有多个重复的， 那么这个变量会取出重复的依赖目标，只保留一份。

### $+ 这个变量很像$^, 也是所有依赖目标的集合。 只是它不去除重复的依赖目标。

### $\* 这个变量表示目标模式中”%”及其之前的部分。

例： 目标是”dir/a.foo.b”, 并且目标的模式是”a.%.b”， 那么表明匹配”a.%”及其之前的部分， 也就是”dir/a.foo”.

## 规则中通配符的使用

### Make支持三个通配符: “\*”, “?”, “[…]”.

### ~字符在文件名中有比较特殊的用途，表示是HOME目录。如“～/test“表示当前用户的$HOME目录下的test目录。 而“~hchen/test“则表示用户hchen的宿主目录下的test目录。

### Makefile中通配符\*的使用

Object=\*.o 通配符并不会展开

Object := $(wildcard \*.o) 通配符展开， 会匹配所有的.o 文件

## 伪目标

Clean:

Rm \*.o temp

我们并不生成clean这个文件。 “伪目标clean”并不是一个文件，它只是一个标签。 又因为为目标后面的prerequisite不存在，因此也无法生成它的依赖关系和决定它是否要执行。只有我们通过显示地指定这个”目标”才能让其生效。

## 不显示命令

通常，make会把其要执行的命令行在命令执行前输出到屏幕上。当我们用”@”字符在命令行前时，这个命令将不会将不会被make显示出来。最具代表性的是，我们用这个功能来像屏幕显示一些信息。如：

@echo compiling some component

如果没有@的话，屏幕输出是：

@echo compiling some component

Compiling some component

但是有了@的话，输出的是：

Compiling some component

## 检查规则

有时候我们不想让我们的makefile中的规则执行起来，我们只想检查下我们的命令，或是执行的序列，于是我们可以使用make命令的下属参数：

-n

--just-print

--dry-run

--recon

不执行makefile， 这些参数指示打印命令，不管目标是否更新， 把规则和连带规则下的命令打印出来，但不执行，这些参数对于我们调式makefile很有用处。

–s: 全面禁止显示命令。

-t

--touch

这个参数的意思是把目标文件的时间更新，但不更改目标文件，也就是说，make假装编译目标，但不是真正的编译目标，只是将目标变成已编译过的状态。

-q

--question

这个参数的行为是找目标的意思，也就是说，如果目标存在，那么其什么也不会输出， 当然也不会执行编译，如果目标不存在，其会打印一条出错信息。

-W <file>

--what-if=<file>

--assume-new=<file>

--new-file=<file>

这个参数需要指定一个文件，一般是源文件或依赖文件，make会根据规则推导来运行依赖于这个文件的命令， 一般来说，可以和-n参数一起使用，来查看这个依赖文件所发生的规则命令。

## Make的参数

-e : 如果指定此参数， 表明系统变量将覆盖makefile中定义的变量。 一般得， 如果makefile中已定义了某个变量， 或者这个变量由makefile命令行带入，那么系统变量的指将被覆盖。但是如果定义了这个值，那么系统变量将不会被覆盖。

-b

-m

忽略和其他版本make的兼容性。

-B

--always-make

认为所有的目标都需要更新（重编译）

-C <dir>

--directory=<dir>

指定读取makefile的目录。如果有多个-C 参数， make的解释是后面的路径以前面的作为相对路径，并以最后的目录作为被指定目录。如”make –C ~hchen/test –C prog” 等价于“make –C ~hchen/test/prog”.

--debug[=<option>]

输出make的调试信息。 它有几种不同的级别可供选择。 如果没有参数， 那就是输出最简单的调试信息。下面是option的取值。

a----也就是all，输出所有的调试信息。

b-----basic。输出不需要重编译的目标。

v-----verbose, 在b选项的级别之上。 输出的信息包括那个makefile被解析，不需要被重编译的依赖文件等。

i------implicit, 输出所以的隐含规则。

j------jobs, 输出执行规则中命令的详细信息， 如命令PID、返回码等。

m-----makefile, 输出make读取makefile， 更新makefile， 执行makefile的信息。

make的参数是”-k” 或者是”—keep-going”, 这个参数的意思是，如果某个规则中的命令出错了，那么终止这个规则，但是继续执行下面的规则。

## Make下的命令执行

当有2条命令时，第二条命令依赖第一条命令的执行结果时，需要将2条命令写在同一行内，且2个命令之间用”;”分隔开。如下

Exec:

Cd /home/hongjyan; pwd

而不能写为

Exec:

Cd /home/hongjyan

Pwd

## 命令出错

每当命令运行完后， make会检测每个命令的返回码， 如果命令返回成功， 那么make会执行下个命令，当所有命令完成后，这个规则算成功完成了，如果一个规则中有一个命令出错了（命令退出码非零），那么make就会终止当前规则，这将有可能终止所有的规则执行。

有时候，我们想即使某个命令出错了，依然继续执行，那么此时可以在这个命令前面加上一个”**-**“号，表明即使命令出错，依然运行下条命令。

还有如果某个规则的target是”.IGNORE”, 那么这个规则中的所有命令都将忽略错误。

如果make的参数是”-k” 或者是”—keep-going”, 这个参数的意思是，如果某个规则中的命令出错了，那么终止这个规则，但是继续执行下面的规则。

## 嵌套执行make

在一些大的工程中，不可能一个make里面写上所有的规则。总是有个总控makefile，然后下面嵌套一些小的makefile，例如：

Subsystem:

Cd subdir && $(MAKE)

注意总控MAKEFILE里的变量可以使用export传递到下级的makefile中，但是不会覆盖下级的makefile中所定义的变量，除非指定了”-e”参数。

如果又不想某些变量传递到下级makefile里的话，那么可以使用unexport <variable>。

如果要传递所有的变量，那么，只要一个export就行了。后面什么也不用跟，表示传递所有的变量。

需要注意的是， 有2个变量: SHELL, MAKEFILES, 这2个变量，不论是否 export了，总是要传递到下层的Makefile中的，特别是MAKEFILES变量， 其中包含了make的参数信息。如果总控Makefile是有make参数或是在上层Makefile中定义了这个变量，那么MAKEFILE变量将会是这些参数，并传递到下层Makefile中，这是一个系统级的环境变量。

但是注意make命令中的-c，-f， -h， -o， -w并不会向下传递参数。

如果不想向下传递参数，那么，可以：

Subsystem：

Cd subdir && $(MAKE) MAKEFLAGS=

-w参数:这是一个在嵌套执行中比较有用的参数， “-w”或是“—print-directory”会在make过程中输出一些信息，让你看到目前的工作目录。如：如果下级目录是”/home/hongjyan”，那么我们使用”make –w“来执行，那么当进入该目录时，我们会看到：

Make: entering directory `/home/hongjyan`，而在完成下层目录时，我们会看到：make：leaving directory `/home/hongjyan`。 当使用-C 参数来指定下层Makefile时， “-w”会被自动打开。如果参数中有“-s”或是”—no-print-directory“，那么， -w总是失效的。

## MAKELEVEL记录当前makefile的调用层数。

## 定义命令包（有点像函数的用法）

命令包相当于make里的函数。他是以define开头的，以endef结尾。如下：

Define run-yacc

Yacc $(firstword $^)

Mv y.tab.c $@

Endef

这里， “run-yacc”是这个命令包的名字，其不能与makefile里的变量重名。在define和endef之间是命令序列。

像使用变量一样使用这个序列, 如下：

Foo.c : foo.y

$(run-yacc)

## 变量重要

### 变量在声明时需要给予初值，而在使用时，需要给变量名前加上$号， 但安全起见，最好用小括号()或者大括号{}, 把变量给包括起来。如果要使用$字符，那么你需要用$$来表示。

变量会在使用它的地方精确展开，就像C/C++中的宏一样，但是MAKEFILE中的变量的值可以随后改变。

### 变量的定义

可以使用=号来定义，也可以用:=来定义。 前者可以使用后面才被定义的值， 后者只能使用已经被定义好了的值。如下：

Y=$(x) bar

X = foo

结果y为foo bar， x为foo。

Y := $(x) bar

X := foo

结果y为bar，x为foo。

### 如何定一个值为空格的变量

Var:= #end of the line

可以使用空格后面加上#表明中间有一个空格。

注意：

dir := /foo/bar/ #end of the line.

如果我们再有一个变量使用到上面变量值${dir}, 如${dir}/file, 那么就完蛋了。

### ?=

FOO ?= bar

这条语句表明如果FOO没有被定义过，那么FOO的值为bar，否则什么都不做。

### 变量的高级用法

#### 变量值替换 方法一：$(var:a=b)或是${var.a=b}

意思var中所有结尾的”a”替换为”b”。这里的结尾意思是“空格“或”结束符“。

示例： foo := a.o b.o c.o

Bar := $(foo:.o=.c)

第二行意思是把$(foo)中所有以”.o”字串结尾全部”.c”, 所以我们的$(bar) 的值就是a.c b.c c.c.

#### 变量替换方法二：”静态模式”，如：

Foo := a.o b.o c.o

Bar := $(foo:%.o=%.c)

Tips：模式是以%开头的。

#### 把变量的值当作另一个变量名

例一：变量作为值用在右边

x =y

y = z

a := $($(x))

在这个例子中， $x的值是y，然后$y的值是z，所以a的值是z。

例二：变量作为值用在右边

X = variable1

Variable2 := Hello

Y = $(subst 1,2,$(x)) #subst的作用是将第3个参数中的所有第一个字符换成第二个字符

Z = y

A := $($($(z)))

$z = y, $y= $(subst 1,2,$(x)), $(subst 1,2,$(x))=variable2, $(variable2)=Hello. 所以a值为Hello。

例三： 函数和条件语句一起使用

Ifdef do\_sort

Func := sort

Else

Func:=strip

Endif

Bar := a d b g q c

Foo := $($(func) $(bar))

如果定义了do\_sort函数， 那么foo值为为a b c d g q. 否则调用strip函数。

例四： 把变量作为值用在左边

Dir = foo

$(dir)\_sources := $(wildcard $(dir)/\*.c)

Define $(dir)\_print

Lpr $($(dir)\_sources)

Endef

这个例子中定义了3个变量, dir, foo\_sources, foo\_print.

#### 追加变量值+=

+=同样适用于makefile， 如果变量之前没有定义过，那么+=会自动变成=,

如果前面有变量定义， 那么+=会继承于前次操作的复制操作符， 也就意味如果前一次是:=, 那么+=会以:=作为其赋值符。

如

Variable := value

Variable += more

等价于：

Variable := value

Variable := $(variable) more

注意：

Variable = value

Variable += more

理论上由于使用=号， 理论上会发生变量的递补归定义。但是别担心makefile会自动处理好这种问题的。

#### Override指示符

如果有变量是通常make的命令行参数设置的， 那么Makefile中对这个变量的赋值会被忽略。 如果你想在Makefile中设置这类参数的值， 那么， 你可以使用override指示符。 其语法是：

Override variable := value

Override variable += more text;

## 环境变量

Make运行时的系统环境变量可以在make开始运行是被载入到makefile中，但是如果makefile已定义了这变量，或是这个变量由make命令行带入， 那么系统的环境变量的值将被覆盖。（如果make指定了-e参数， 那么， 系统的环境变量将覆盖makefile中定义的变量）。

因此，如果我们在环境变量中设置了CFLAGS环境变量， 那么我们就可以在所有的makefile中使用这个变量了。这对于我们使用同一的编译参数有比较大的好处。 如果没有使用-e这个参数，且makefile中定义了CFLAGS， 那么则会使用makefile中的这个变量。环境变量和makefile里面定义的变量有点像全局变量和局部变量的这个特性。

默认情况下，只有通过命令行设置的变量会被传递，而定义在文件中的变量入股要想下层传递的话，则需要使用export关键字来声明。

当然，不推荐把许多的变量定义在环境变量里， 这样，在我们执行不同的makefile时，用有同一套系统变量，这可能会带来更多的麻烦。

## 目标变量

前面我们所讲的在makefile中定义的变量都是”全局变量”, 在整个文件， 我们都可以访问这些变量。而如”$<”, “$@”这种自动化变量就属于”规则型变量”, 这种变量的值依赖于规则的目标和依赖目标的定义。

我们同样可以为某个目标设置局部变量， 这种变量被称为“Target-specific Variable”, 它可以和”全局变量”同名， 因为它的作用范围只在这条规则以及连带规则中， 所以其只在作用范围内有效，而不会影响规则链以外的全局变量的值。其语法是:

Target： variable-assignment；

Tartget: override <variable-assignment>

<variable-assignment>可以是前面讲过的各种赋值表达式， 如”=”、“：=”、“+=”、“？=”。第二个语法是针对于make命令行带入的变量，或系统环境变量。语法如下:

Prog: CFLAGS = -g

Prog: prog.o foo.o bar.o

$(CC) $(FLAGS) prog.o foo.o bar.o

Prog.o : prog.c

$(CC) $(CFLAGS) prog.c

Foo.o: foo.c

$(CC) $(CFLAGS) foo.c

Bar.o : bar.c

$(CC) $(CFLAGS) bar.c

在prog这个target里面设置了CFLAG这个变量， 这个变量会作用到由这个目标引发的所有的规则（包括prog.o, foo.o bar.o）中去， 这个目标引发的所有的规则里， $(CFLAG)的值都是-g。

## 模式变量

在GNU的make中， 还支持模式变量（pattern-specific variable）, 模式变量的好处是： 我们可以定一个模式，可以把变量定义在符合这种模式的所有目标上。

我们知道， make的“模式”一般都至少含一个”%”的， 所以，我们可以以如下方式给所有的以.o结尾的目标定义目标变量：

%.o : CFLAGS = -O

同样， 模式变量的语法和”目标变量“一样：

<pattern> : <variable-assignment>

<pattern> : override <variable-assignment>

## 使用条件判断

<conditional-directive>;

<text-if-true>;

Else

<text-if-false>

Endif

其中<conditional-directive>, 表示条件关键字， 如ifeq， 这个关键字有四个。

第一个是ifeq，第二个是ifneq， 第三个是ifdef， 第四个是ifndef。

示例：

Libs\_for\_gcc = -lgnu

Normal\_libs =

Foo: $(objects)

Ifeq ($(CC), gcc)

$(CC) –o foo $(objects) $(libs\_for\_gcc)

Else

$(CC) –o foo $(objets) $(normal\_libs)

Endif

特别注意的是， make在读取makefile时就计算条件表达式的值， 并根据条件表达式的值来选择语句， 所以做好不要将自动化自动化变量如$@放入到条件表达式中， 因为自动化变量是在运行时才有的。

## 函数调用

$(<function> <argument1>, <argument2>,….)

${<function> <argument1>, <argument2>,….}

函数调用以“$”开头， 以圆括号或者花括号把函数名和参数括起来。

且函数中可以使用变量，如果函数参数中有变量的话，为了风格的统一， 此时最好使用圆括号。如下$(subst a,b,$(x))比${subst a,b, $(x)}要清晰。

例：将空格转换为逗号

Comma :=,

Empty :=

Space := $(empty)

Foo := a b c

Bar := $(subst $(space), $(comma), $(foo))

所以$(bar)的值是“a, b, c”.

## 字符串处理函数

### 替换函数$(subst <from>, <to>, <text>)

字符串替换函数， 将text里的from全部替换为to。

### 模式替换函数 $(patsubst <pattern>, <replacement>, <text>)

将text里面pattern部分全部替换为replacement的部分。注意pattern和replacement都可以是一种模式。

示例:

$(patsubst %.c, %o, x.c.c bar.c)

把字串”x.c.c bar.c”符合模式[%.c]的单词替换成[%.o], 应为%表示任意长度的字串， 所以返回结果是”x.c.o bar.o”。 这和前面的变量章节的使用静态模式进行变量替换有些相似。如上，也可以用

object = x.c.c bar.c

$(object:.c=.o)

来替换。

### $(strip <string>)，去掉字符开头或者结尾的空格字符

### $(findstring <find>, <in>) 查找字符串函数

在in里面查找find， 如果找到返回find，否则，返回空字串。

### $(filter-out <pattern>, <text>)反过滤函数

以pattern模式过滤text，去除符合模式的单词，可以有多个模式。返回不符合模式的字串。

Objects = main1.o foo.o main2.o bar.o

Mains = main1.o main2.o

$(filter-out $(mains),$(objects))返回值是”foo.o bar.o”.

### $(sort <list>)排序函数

返回排序后的list。备注sort会去掉list里面的重复函数。

### $(word n, text)

取单词函数， 取出text里面的第n个单词。 返回字符串text中第n个单词（注意不是字符），如果n比text中的单词数要大，那么返回空字符串。如$(word 2, foo bar baz)返回值是bar。

### $(wordlist n, m, text) 取单词串函数

从字符串text中，取出n到m之间的单词串。如果n比单词数要大，那么返回空字符串。如果m要大于text中的单词数，那么返回从n开始到text结束的单词串。如$(wordlist 2, 3, foo bar baz)返回值是bar baz。

### $(words text)单词个数统计

统计text中的单词个数。 如$(word, foo bar baz)， 返回的是3.

### $(firstword text) 取首单词函数

取字符串text中的第一个单词， 如$(firstword foo bar)返回值是foo。备注：这个函数可以用word函数来实现$(word 1, text).

### 一个综合使用的例子

Override CFLAGS += $(patsubst %, -I%, $(subst :, , $(VPATH)))

如果VPATH的值为src:../headers, 那么$(patsubst %, -I%, $(subst :, , $(VPATH)))将返回”-Isrc –I../headers”, 这正是cc或者gcc搜索头文件路径的参数。

## 文件名操作函数

### $(dir <name>) 取目录函数

从文件名序列names中取出目录部分。目录部分是指最后一个反斜杠/之前的部分。如果没有反斜杠，那么返回”./”。如$(dir src/foo.c hacks)返回值是”src/ ./”。

### $(notdir <names>) 取文件函数

从文件名序列names中取出非目录部分。非目录的部分指最一个反斜杠”/”之后的部分。如果没有/，那么直接返回所有值。$(notdir src/foo.c hacks)返回值是”foo.c hacks”。

### $(suffix <names>) 取后缀函数

返回文件的后缀，若没有后缀，那么返回空字符串。如$(suffix src/foo.c src-1.0/bar.c hacks)返回值是“.c .c”。

### $(basename <names>)取前缀函数

返回names的前缀序列，如果文件没有前缀，则返回空字符串。如$(basename src/foo.c src-l.0/bar.c hacks)返回值是“src/foo src-1.0/bar hacks)

### $(addsuffic <suffic>, <names>) 加后缀函数

功能是把后缀加到names中的每个单词后面。返回加过后缀的文件名序列。 如$(addsuffix .c, foo bar)返回值是”foo.c bar.c”。

### $(addprefix <prefix>, <names>)加前缀函数

把前缀prefix加到names中每个单词的前面。返回操作后的结果。 如$(addprefix src/, foo bar)返回值是“src/foo src/bar”。

### $(join <list1>, <list2>) 连接函数join

功能是将list2中的单词对应加到list1的单词后面。如果list1的单词个数比list2中的多， 那么list1中多出的单词将保持原样，如果list2多出来的单词将被复制到list1中。 返回连接后的字符串。

如$(join aaa bbb, 111 222 333)， 返回值是”aaa111, bbb222, 333”.

## Foreach函数

$(foreach <var>, <list>, <text>) 这个函数将list中的值逐一取出给var， 然后执行text里面的语句，得到一个字符串。循环过程中，text的所返回的每个字符串会以空格分隔，当最后整个循环结束时， text所返回的每个字符串所组成的整个字符串（以空格分隔）将会是foreach函数的返回值。 var一般是一个变量名， list是一个列表， 而text是处理var值的某段语句。

例：

names := a b c d

files := $(foreach n, $(names), $(n).o)

上面例子中$(files)的值是”a.o b.o c.o d.o”。

注意foreach中var参数是一个临时局部变量， foreach函数执行完毕后， 参数var的变量将不再作用， 起作用只在foreach函数当中。

## If 函数

$(if <condition>, <then-part>, <else-part>)

也可以是

$(if <condition>, <then-part>)。此时如果条件为假，又没有else part，那么将返回空字符串。

## Call 函数

$(call <expression>, <param1>, <param2>, <param3>)

Call函数的作用是用来创建一个新函数， 函数体是expression， 变量名为定死的$1, $2, $3. 分别对应param1， param2, param3。 如下

Foo = $(call $1 $2, a, b) 那么foo的值为a b

## Origin函数

Origin函数并不像其他函数， 它并不操作变量的值， 它只是告诉你你的这个变量是从哪里来的。 其语法是：

$(origin <variable>)

注意<variable>是变量的名字，不应该是应用。所以最好不要在<variable>中使用$字符。 Origin函数会以其返回值来告诉你这个变量的“出生情况“。下面，是origin函数的返回值：

“undefined” 如果variable从来没有定义过，origin函数返回这个值”undefined”

“default” 如果variable是一个默认的定义，如”CC”这个变量，这种变量我们在后面讲述。

“environment” 如果variable是一个环境变量。

“file” 如果variable这个变量被定义在makefile中的。

“command line” 这个变量是被命令行定义的。

“override” 这个变量是被override指示符重新定义的。

“automatic” 这个变量是命令运行中的自动化变量。

## Shell函数重要

Contents := $(shell cat foo) 等同于contents = `cat foo`

## 控制make的函数

$(error <text>) 输出text错误信息，并且停止make的执行

$(warning <text>) 输出text警告信息， 而make继续执行。

## Make的退出码

0----表示成功执行

1----如果make运行出现任何错误，其返回1

2----如果使用了make –q选项，并且make使一些目标不需要更新，那么返回2。

## GNU make命令的寻找顺序

在当前目录下依次找三个文件“GNUmakefile“， “makefile“， “Makefile“这三个文件。

## 指定目标

Make命令可以将makefile中任何一个target作为第二个参数， 来作为目标进行编译。 MAKECMDGOALS这个变量记录存放的是指定的终极目标的列表。如果在命令行上， 没有值指定目标，那么这个变量的值为空。

如下：

.PHONY : all

All : prog1 prog2 prog3 prog4

可以使用make all 命令来编译所有的目标，也可以使用make prog2来单独编译目标prog2。

参照GNU规则来书写我们自己的makefile中的目标。

All: 这个伪目标是所有目标的目标，其功能一般是编译所有的目标。

Clean: 这个伪目标功能是删除所有被make创建的文件。

Install： 这个伪目标是安装已编译好的程序，其实就是把目标的执行文件拷贝到指定的目标中去。

Print: 列出改变过的源文件。

Tar: 将源程序打包备份，也就是一个tar文件。

Dist：创建一个压缩文件， 一般是将tar文件压成z文件。或是gz文件。

TAGS: 更新所有的目标，以备完整地重新编译使用。

Check和test： 这2个伪目标一般用来测试makefile的流程。

## 隐含规则搜索算法

比如我们有一个目标叫 T。下面是搜索目标T的规则的算法。请注意，在下面，我们没有提到后缀规则，原因是，所有的后缀规则在Makefile被载入内存时，会被转换成模式规则。如果目标是"archive(member)"的函数库文件模式，那么这个算法会被运行两次，第一次是找目标T，如果没有找到的话，那么进入第二次，第二次会把"member"当作T来搜索。   
  
1、把T的目录部分分离出来。叫D，而剩余部分叫N。（如：如果T是"src/foo.o"，那么，D就是"src/"，N就是"foo.o"）   
  
2、创建所有匹配于T或是N的模式规则列表。   
  
3、如果在模式规则列表中有匹配所有文件的模式，如"%"，那么从列表中移除其它的模式。   
  
4、移除列表中没有命令的规则。   
  
5、对于第一个在列表中的模式规则：   
    1）推导其"茎"S，S应该是T或是N匹配于模式中"%"非空的部分。   
    2）计算依赖文件。把依赖文件中的"%"都替换成"茎"S。如果目标模式中没有包含斜框字符，而把D加在第一个依赖文件的开头。   
3）测试是否所有的依赖文件都存在或是理当存在。（如果有一个文件被定义成另外一个规则的目标文件，或者是一个显式规则的依赖文件，那么这个文件就叫"理当存在"）   
    4）如果所有的依赖文件存在或是理当存在，或是就没有依赖文件。那么这条规则将被采用，退出该算法。   
  
6、如果经过第5步，没有模式规则被找到，那么就做更进一步的搜索。对于存在于列表中的第一个模式规则：   
    1）如果规则是终止规则，那就忽略它，继续下一条模式规则。   
2）计算依赖文件。（同第5步）   
3）测试所有的依赖文件是否存在或是理当存在。   
4）对于不存在的依赖文件，递归调用这个算法查找他是否可以被隐含规则找到。   
5）如果所有的依赖文件存在或是理当存在，或是就根本没有依赖文件。那么这条规则被采用，退出该算法。  
  
7、如果没有隐含规则可以使用，查看".DEFAULT"规则，如果有，采用，把".DEFAULT"的命令给T使用。   
  
一旦规则被找到，就会执行其相当的命令，而此时，我们的自动化变量的值才会生成。

## 函数库文件也就是对object文件的打包文件。在Unix下， 一般是有命令”ar”来完成打包工作的。

## 函数库文件的成员

一个库文件由多个文件组成。你可以以如下格式指定函数库文件及其组成:

Archive(member)

这不是一个命令，而是一个目标和依赖的定义。一般来说，这种用法基本上就是为了”ar”命令服务的。如：

Foolib(hack.o): hack.o

Ar cr foolib hack..o

如果要指定多个member，就是用空格将它们分开， 如：

Foolib(hack.o kludge.o)

还可以使用shell的文件通配符来定义，如：

Foolib(\*.o)

## 函数库成员的隐含规则

当make搜素一个目标的隐含规则是， 一个特殊的特性是，如果这个目标是”a(m)”形成的， 他会以m为目标，利用用户自定义的规则或者隐式规则来生成m, 然后调用ar命令将m打包生成库a。如下

Make foo.a(bar.o)

这个等同于

bar.o :bar.c

cc –c bar.c –o bar.o

ar r foo.a bar.o

rm –f bar.o

## 函数库文件的注意事项

在进行函数库文件打包文件生成时， 小心使用make的并行机制(-j 参数)。 如果多个ar命令在同一时间运行在同一个函数库文件打包文件上， 可能损害这个库文件。 所以， 在make未来的版本中， 所以尽量不要使用-j 参数。

## Makefile里面如何进行算术运算

JPI=4

JPJ=2

FOO=$(shell echo $(JPI)\\*$(JPJ) | bc)

all:

echo $(FOO)

少了bc不行， 可以echo出来，但是如果将${FOO}用在别的变量里面为4\*2

# 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>

# 汇编基础

## 编译一个汇编

将文件保存为xxx.asm.

* Nasm –f elf xxx.asm //xxx.o 文件会产生
* Ld –m elf\_i386 –s –o xxx xxx.o //可执行文件xxx会产生。

## The data section

section.data

The **data** section is used for declaring initialized data or constants. This data does not change at runtime. You can declare various constant values, file names, or buffer size, etc., in this section.

## The bss Section

The **bss** section is used for declaring variables. The syntax for declaring bss section is −

section.bss

## The text section

The **text** section is used for keeping the actual code. This section must begin with the declaration **global \_start**, which tells the kernel where the program execution begins.

The syntax for declaring text section is −

section.text

global \_start

\_start:

## Comments

Assembly language comment begins with a semicolon (;). It may contain any printable character including blank. It can appear on a line by itself, like −

; This program displays a message on screen

or, on the same line along with an instruction, like −

add eax, ebx ; adds ebx to eax

## Statements

* Executable instructions or instructions,
* Assembler directives or pseudo-ops, and
* Macros.

The **executable instructions** or simply **instructions** tell the processor what to do. Each instruction consists of an **operation code** (opcode). Each executable instruction generates one machine language instruction.

The **assembler directives** or **pseudo-ops** tell the assembler about the various aspects of the assembly process. These are non-executable and do not generate machine language instructions.

**Macros** are basically a text substitution mechanism.

## Statement syntax

Assembly language statements are entered one statement per line. Each statement follows the following format −

[label] mnemonic [operands] [;comment]

The fields in the square brackets are optional. A basic instruction has two parts, the first one is the name of the instruction (or the mnemonic), which is to be executed, and the second are the operands or the parameters of the command.

Following are some examples of typical assembly language statements −

INC COUNT ; Increment the memory variable COUNT

MOV TOTAL, 48 ; Transfer the value 48 in the

; memory variable TOTAL

ADD AH, BH ; Add the content of the

; BH register into the AH register

AND MASK1, 128 ; Perform AND operation on the

; variable MASK1 and 128

ADD MARKS, 10 ; Add 10 to the variable MARKS

MOV AL, 10 ; Transfer the value 10 to the AL register

## 进程寄存器

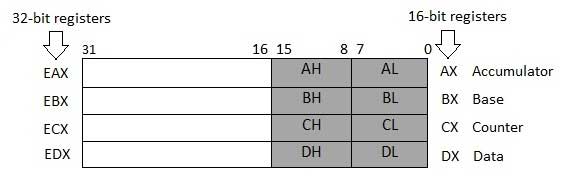
Process registers包含三大类： General register， Control Register， Segment register。

## General register

General register又被分为三类： data registers, Pointer register, index register.

### Data register

其中data register有32位的4种：如下。如果32位不被全部使用的话， 低16位被用来作为4个16-bit 的data-register。而16位又可以按8位划分为8个8bit的data-register如下：



**AX is the primary accumulator 主累加器**; it is used in input/output and most arithmetic instructions. For example, in multiplication operation, one operand is stored in EAX or AX or AL register according to the size of the operand.

**BX is known as the base register基础寄存器**, as it could be used in indexed addressing.

**CX is known as the count register计数寄存器**, as the ECX, CX registers store the loop count in iterative operations.

**DX is known as the data register数据寄存器，DX可以和AX一起使用用于对大值的乘/除操作**. It is also used in input/output operations. It is also used with AX register along with DX for multiply and divide operations involving large values.

### 指针寄存器Pointer registers

包含IP(instruction pointer)， SP(stack pointer), BP(Base Pointer). 32位的前面加个E(extended).

IP: 存储下一个instruction 的offset。 配合CS register（code segement register）使用，得到当前instruction的complete地址。

SP: 用于表示stack中数据的offset。 配合SS register使用，得到于stack中的data的complete地址

BP: 用于寻找传给函数的参数地址， 配合 SS register（stack segement register）使用， 得到函数实参的完整地址。还可以下面的DI 和SI配合使用 for special addressing。

### 索引寄存器 Index registers

SI(Source Index)： − It is used as source index for string operations.

DI(Destination index)： − It is used as destination index for string operations.

不知道上面2个的区别。

## Segment register

CS register（code segment register）： stores the starting address of the code segment

DS register (data segment register): stores the starting address of the data segment

SS register (stack segement register): stores the the starting address of the stack.

以上的segment register存储的都是“起始地址”， 要获取data或instruction的精确地址， +offset就可以了。

## 取array地址（direct-offset Addressing）

## Indirect Memory Addressing

This addressing mode utilizes the computer's ability of *Segment:Offset* addressing. Generally, the base registers EBX, EBP (or BX, BP) and the index registers (DI, SI), coded within square brackets for memory references, are used for this purpose.

Indirect addressing is generally used for variables containing several elements like, arrays. Starting address of the array is stored in, say, the EBX register.

The following code snippet shows how to access different elements of the variable.

MY\_TABLE TIMES 10 DW 0 ; Allocates 10 words (2 bytes) each initialized to 0

MOV EBX, [MY\_TABLE] ; Effective Address of MY\_TABLE in EBX。 此处就取了数组MY\_TABLE的地址。

MOV [EBX], 110 ; MY\_TABLE[0] = 110

ADD EBX, 2 ; EBX = EBX +2 通过EBX， 再+2， 间接取到了MY\_TABLE[1] （因为数组是DW即2 bytes）的地址。

MOV [EBX], 123 ; MY\_TABLE[1] = 123

## 方括号[ ]

[X], 如果X表示寄存器如[BX]，似乎表明解引用（类似C中\*），即BX的值

[X], 如果X表示普通变量如[name]， 似乎表明引用(类似C中的&), 即取name的地址。对于普通变量， assembler将offset value与变量名相关联起来。

## 数据类型/define directive

Directive Purpose Storage Space

DB Define Byte allocates 1 byte

DW Define Word allocates 2 bytes

DD Define Doubleword allocates 4 bytes

DQ Define Quadword allocates 8 bytes

DT Define Ten Bytes allocates 10 bytes

## 申请内存大小

Directive Purpose

RESB Reserve a Byte

RESW Reserve a Word

RESD Reserve a Doubleword

RESQ Reserve a Quadword

REST Reserve a Ten Bytes

segment .bss

num1 resb 2

num2 resb 2

res resb 1

## TIMES 声明及初始化数组

marks TIMES 9 DW 0 ;declare 9 elelment DW array named "marks" and initialize each element as 0.

## equ/常量标志符

CONSTANT\_NAME EQU expression

例子： TOTAL\_STUDENTS equ 50

## %assign。赋值操作符，注意只能赋数值

%assign TOTAL 10 ;赋值TOTAL 为10

%assign TOTAL 20 ;重新赋值TOTAL为20

## %define。 类型#define in c

所以可以%define PTR [EBP+4]

%define PTR [EBP+5] //允许重定义

## Linux系统调用

调用过程：

" Store the arguments to the system call in the registers EBX, ECX, etc.

" Put the system call number in the EAX register.

" Call the relevant interrupt (80h).

" The result is usually returned in the EAX register.

代码如下：

mov edx,4 ; message length

mov ecx,msg ; message to write

mov ebx,1 ; file descriptor (stdout)

mov eax,4 ; system call number (sys\_write)

int 0x80 ; call kernel

有6个寄存器用于存储系统调用的参数。 These are the EBX, ECX, EDX, ESI, EDI, and EBP. These registers take the consecutive arguments, starting with the EBX register. If there are more than six arguments, then the memory location of the first argument is stored in the EBX register. （只有1个，或2个的参数的函数， 存储参数， 也是从EBX开始的啊， 如下sys\_exit的唯一一个参数就存储在ebx的啊）。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **%eax** | **Name** | **%ebx** | **%ecx** | **%edx** | **%esx** | **%edi** |
| 1 | sys\_exit | int | - | - | - | - |
| 2 | sys\_fork | struct pt\_regs | - | - | - | - |
| 3 | sys\_read | unsigned int | char \* | size\_t | - | - |
| 4 | sys\_write | unsigned int | const char \* | size\_t | - | - |
| 5 | sys\_open | const char \* | int | int | - | - |
| 6 | sys\_close | unsigned int | - | - | - | - |

## INC/DEC

INC EBX ; Increments 32-bit register

INC DL ; Increments 8-bit register

INC [count] ; Increments the count variable //似乎[普通变量]， 此时取得是普通变量的值，不是取普通变量的地址了。

dec byte [ESI]

inc word [ebx]

## MUL/IMUL

MUL: 无符号的

IMUL: 有符号的

# 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.

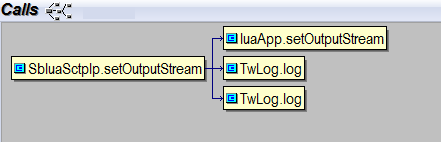
For example, functions that are called by the current function.

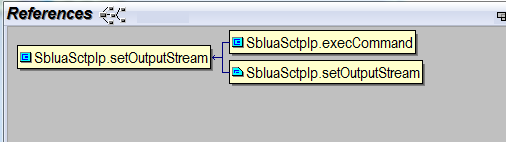
**• References** – show what other symbols refer to the current symbol.

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

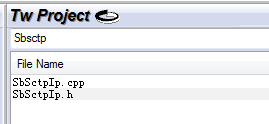
## 在project中查找某symbol

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

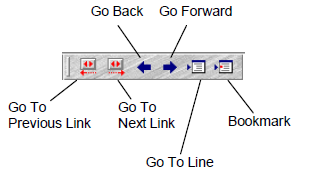


you type \*.c and press

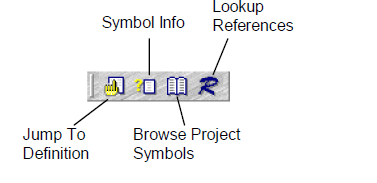
Enter, you will see all \*.c files in your project, regardless of directory. To remove

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

## 设置标签



## 符号图标



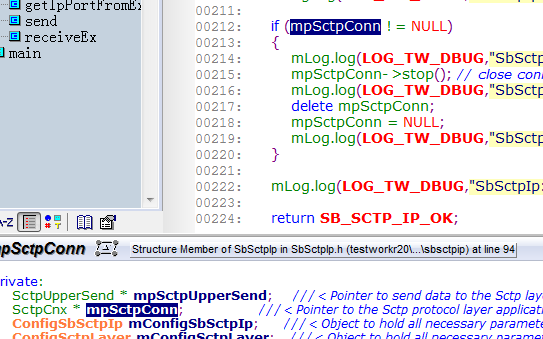
第一个是右键的快捷方式

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

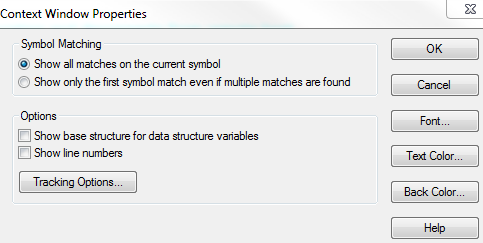
 最后一个会显示上一个search result。

## 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。

## Eclipse技巧

### 显示line number

Window → Preferences → General → Editors → Text Editors → Show line numbers.

# 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 或者 break TwLog.cpp:121 //在TwLog.cpp的121行给break

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

## TW设置断点

先把bearerChannel给建起来， pe –ef | grep Sb 获得BearerChannel binaray文件的pid，gdb🡪attach $pid-🡪break TwLog.cpp:121, 然后跑下面的TP， 如果程序到了break 那个点，就会停住。

## 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

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

**\*  soft  core  unlimited**

2. 重启

**3.**

**[tester@mango coredir]$ sudo service abrtd stop**

**[tester@mango coredir]$ sudo chkconfig abrtd off //关闭 automatic bug report service**

**[tester@mango coredir]$ sudo service abrt-ccpp stop**

**[tester@mango coredir]$ sudo chkconfig abrt-ccpp off //关闭abrt-ccpp**

**[tester@mango coredir]$ sudo service abrt-oops stop**

**[tester@mango coredir]$ sudo chkconfig abrt-oops off //同上**

**4.mkdir /tmp/coredir; chmod 777 /tmp/coredir;**

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

6. 向文件/etc/sysctl.conf里加入： kernel.core\_pattern = /tmp/coredir/coredump\_%e\_%p

注意：带有debug信息的binary文件产生的coredump里的信息才是有用的， 否则你不知道哪个coredump产生的具体原因

## 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。

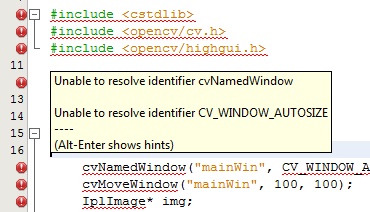
BASE指你本地机器上的最新版本。

## Netbean里如果出现“unable to resolve identifier”错误，但编译没问题

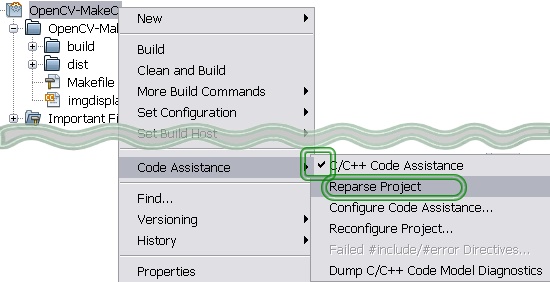
Here's an excerpt from my answer to another question.

**Unresolved Identifier**

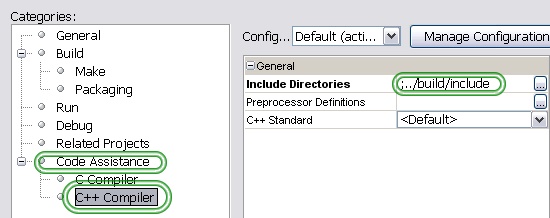
If the source of the .cpp file looks like this



Click with the right mouse button on your project.  
Check C/C++ Code As...  
Run Reparse Project.



If that is not enough.  
Go to Project Properties  
Fill in the Include input field as described.



Set the include path correct.

I hope that can help you.

## 将JAVA中的exceptiion backtrace打印到文件

Import java.io.PrintStream;

PrintStream ps = new PrintStream("/home/tester/TpEntityAction.txt");

e.printStackTrace(ps);