C++内存与堆栈机制

在C++中，内存分成5个区，他们分别是堆、栈、自由存储区、全局/静态存储区和常量存储区。

**栈（堆栈）**，就是那些由编译器在需要的时候分配，在不需要的时候自动清除的变量的存储区。里面的变量通常是局部变量、函数参数等。

**堆**，就是那些由new分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个new就要对应一个delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。

**自由存储区**，就是那些由malloc等分配的内存块，他和堆是十分相似的，  
不过它是用free来结束自己的生命的。

**全局/静态存储区**，全局变量和静态变量被分配到同一块内存中，在以前的  
C语言中，全局变量又分为初始化的和未初始化的（初始化的全局变量和静态变量在一块区域，未初始化的全局变量与静态变量在相邻的另一块区域，同时未被初始化的对象存储区可以通过void\*来访问和操纵，程序结束后由系统自行释放），在C++里面没有这个区分了，他们共同占用同一块内存区。

**常量存储区**，这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改（当然，你要通过非正当手段也可以修改，而且方法很多）

**明确区分堆与栈**

我们举一个例子：

void f() { int\* p=new int[5]; }

这条短短的一句话就包含了堆与栈，看到new，我们首先就应该想到，我们

分配了一块堆内存，那么指针p呢？他分配的是一块栈内存，所以这句话的意思

就是：在栈内存中存放了一个指向一块堆内存的指针p。在程序会先确定在堆中

分配内存的大小，然后调用operator new分配内存，然后返回这块内存的首地

址，放入栈中。

**堆和栈究竟有什么区别？**

**1、管理方式不同；**

**2、空间大小不同；**

**3、能否产生碎片不同；**

**4、生长方向不同；**

**5、分配方式不同；**

**6、分配效率不同；**

**管理方式**：对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生memory leak。

**空间大小**：一般来讲在32位系统下，堆内存可以达到4G的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定的空间大小的，例如，在VC6下面，默认的栈空间大小是1M（好像是，记不清楚了）。当然，我们可以修改：打开工程，依次操作菜单如下：Project->Setting->Link，在Category 中选中Output，然后在Reserve中设定堆栈的最大值和commit。  
注意：reserve最小值为4Byte；commit是保留在虚拟内存的页文件里面，它设  
置的较大会使栈开辟较大的值，可能增加内存的开销和启动时间。

**碎片问题：**对于堆来讲，频繁的new/delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在他上面的后进的栈内容已经被弹出，详细的可以参考数据结构。

**生长方向：**对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。  
**分配方式：**堆都是动态分配的，没有静态分配的堆。栈有2种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配  
由alloca函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由  
编译器进行释放，无需我们手工实现。  
**分配效率：**栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是C/C++函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。从这里我们可以看到，堆和栈相比，由于大量new/delete的使用，容易造成大量的内存碎片；由于没有专门的系统支持，效率很低；由于可能引发用户态和核心态的切换，内存的申请，代价变得更加昂贵。所以栈在程序中是应用最广泛的，就算是函数的调用也利用栈去完成，函数调用过程中的参数，返回地址，EBP和局部变量都采用栈的方式存放。所以，我们推荐大家尽量用栈，而不是用堆。虽然栈有如此众多的好处，但是由于和堆相比不是那么灵活，有时候分配大量的内存空间，还是用堆好一些。  
无论是堆还是栈，都要防止越界现象的发生（除非你是故意使其越界），因为越界的结果要么是程序崩溃，要么是摧毁程序的堆、栈结构，产生以想不到的结果,就算是在你的程序运行过程中，没有发生上面的问题，你还是要小心，说不定什么时候就崩掉，那时候debug可是相当困难的：）

例子：

int a = 0; 全局初始化区  
char \*p1; 全局未初始化区  
main()  
{  
int b; //栈  
char s[] = “abc”; //栈  
char \*p2; //栈  
char \*p3 = “123456”; //123456在常量区，p3在栈上。  
static int c =0； //全局（静态）初始化区  
p1 = (char \*)malloc(10);  
p2 = (char \*)malloc(20);  
//分配得来得10和20字节的区域就在堆区。  
strcpy(p1, “123456”); //123456放在常量区，编译器可能会将它与p3所指向的"123456"优化成一个地方。  
}

C++反向迭代器

以string为例：

string body = “hello world”;

string::iterator iter = body.begin();

string::iterator iter1 = body.end() - 1;

cout << \*iter << endl;//输出h

cout << \*iter1 << endl;//输出d

cout << "倒序输出：" << endl;

for (string::reverse\_iterator it = body.rbegin(); it != body.rend(); it++)

cout << \*it << endl;//倒序输出“hello world”

三元表达式解析器

一道leetcode上的vip题，实现并记录一下。

题目：

Given a string representing arbitrarily nested ternary expressions, calculate the result of the expression. You can always assume that the given expression is valid and only consists of digits 0-9, ?, :, T and F (T and Frepresent True and False respectively).

Note:

The length of the given string is ≤ 10000.

Each number will contain only one digit.

The conditional expressions group right-to-left (as usual in most languages).

The condition will always be either T or F. That is, the condition will never be a digit.

The result of the expression will always evaluate to either a digit 0-9, T or F.

Input：“T?2:3”

Output：“2”

Input：“F?1:T?4:5”

Output：“4”

Input：“T?T?F:5:3”

Output：“F”

stack<char> body;

string input = "";

bool flag = true;

cin >> input;

string::iterator iter = input.begin();

while (iter != input.end()) {

body.push(\*iter);

iter++;

}

input = "";

while (!body.empty()) {

if (body.top() != '?') {

input += body.top();

body.pop();

}

else {

body.pop();//先将问号弹出

if (body.top() == 'T') {

body.pop();//弹出符号T

for (string::reverse\_iterator it = input.rbegin(); it != input.rend(); it++) {

if (\*it == ':' && flag) {

flag = false;

it += 2;

if (it > input.rbegin())

break;

continue;

}

body.push(\*it);

}

}

else {

body.pop();//弹出符号F

for (string::reverse\_iterator it = input.rbegin(); it != input.rend(); it++) {

if (\*it == ':' && flag) {

flag = false;

continue;

}

if (!flag) {

body.push(\*it);

}

}

}

flag = true;

input = "";

}

}

cout << input << endl;

C++中的0、NULL、nullptr

本质上讲：

1) 0是int型的字面值常量

2) NULL 是预处理变量，定义在 cstdlib 中，其值是0

3) nullptr 是 nullptr\_t 类型的字面值。

cstdlib 中 NULL 的定义

#ifdef \_\_cplusplus

#define NULL 0

#else

#define NULL ((void \*)0)

#endif

之所以这样定义，是因为在C语言中，允许void\*类型隐式转换为任意指针类型，而C++中不允许这样的强制类型转换，但是可以为任意类型的指针赋0值，因此，在C++中将NULL定义为0。

为了避免“野指针”（即指针在首次使用之前没有进行初始化）的出现，我们声明一个指针后最好马上对其进行初始化操作。如果暂时不明确该指针指向哪个变量，则需要赋予NULL值。除了NULL之外，C++11新标准中又引入了nullptr来声明一个“空指针”，之所以引入nullptr，是因为NULL在C++中不完全兼容C，C中NULL的定义为：

#define NULL ((void \*)0)

也就是说NULL实质上是一个void\*指针。NULL在C++中不完全兼容C的原因和C++的函数重载机制有关。如果C++让NULL也支持void \*的隐式类型转换，这样编译器就不知道应该调用哪一个函数。例子如下：

void Func(char \*);

void Func(int);

int main()

{

Func(NULL);

}

如果C++让NULL也支持void\*的隐式类型转换，这样编译器就不知道该调用哪个函数，C++把NULL定义为0，解决了函数重载后的函数匹配问题，但是又引入了另一个问题，同样是这一段代码，由于我们经常使用NULL表示空指针，所以从程序员的角度来看，Func（NULL）应该调用的是Func（char \*）但实际上NULL的值是0，所以调用了Func（int）。nullptr关键字真是为了解决这个问题而引入的。另外我们还有注意到NULL只是一个宏定义，而nullptr是一个C++关键字。

nullptr的使用：nullptr关键字用于标识指针，是std::nullptr\_t类型的（constexpr）变量。它可以转换成任何指针类型和bool布尔类型（主要是为了兼容普通指针可以作为条件判断语句的写法），但是不能被转换为整数：

char \*p1 = nullptr; // 正确

int \*p2 = nullptr; // 正确

bool b = nullptr; // 正确. if(b)判断为false

int a = nullptr; // 错误

单调栈算法模板

单调栈是一种特殊的栈，特殊之处在于栈内的元素都保持一个单调性，可能为单调递增，也可能为单调递减。以Next Greater Number问题为例，假设所遍历的数组是环形数组：

vector<int> nextGreaterElements(vector<int>& nums) {

int n = nums.size();

vector<int> res(n); // 存放结果

stack<int> s;

// 假装这个数组长度翻倍了

for (int i = 2 \* n - 1; i >= 0; i--) {

while (!s.empty() && s.top() <= nums[i % n])

s.pop();

res[i % n] = s.empty() ? -1 : s.top();

s.push(nums[i % n]);

}

return res;

}

动态规划基本思想与举例

例题：给定一个整数数组prices，其中第i个元素代表了第i天股票的价格；非负整数fee代表了交易股票的手续费用。

可以无限次的完成交易，但是每次交易都需要付手续费。如果已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

示例 :

输入: prices = [1, 3, 2, 8, 4, 9], fee = 2

输出: 8

解释: 能够达到的最大利润:

在此处买入 prices[0] = 1

在此处卖出 prices[3] = 8

在此处买入 prices[4] = 4

在此处卖出 prices[5] = 9

总利润: ((8 - 1) - 2) + ((9 - 4) - 2) = 8.

0 < prices.length <= 50000.

0 < prices[i] < 50000.

0 <= fee < 50000.

这类问题要用动态规划的思想进行穷举，本题利用[状态]进行穷举。具体到每一天，看看总共有几种可能的状态，穷举的目的是根据对应的[选择]更新状态：

for 状态1 in 状态1的所有取值：

for 状态2 in 状态2的所有取值：

for ...

dp[状态1][状态2][...] = 择优(选择1，选择2...)

比如这个问题，每天有三种选择[选择]:买入、卖出、无操作，我们用buy，sell，rest表示这三种选择。但问题是，并不是每天都可以任意选择这三种选择的，因为sell必须在buy后。buy必须在sell之前。那么rest操作还应该分两种状态。一种是buy之后的rest(持有了股票)，一种是sell之后的rest(没有持有股票)。而且别忘了，还有交易次数k的说法，就是说buy只能在k>0的前提下操作。

所以这个问题的[状态]有三个，第一个是天数，第二个是允许交易的最大次数，第三个是当前持有状态(即之前说的rest的状态，我们不妨把1表示持有，0表示没有持有)。然后我们用一个三维数组就可以装下这几种状态的全部组合：

dp[i][k][0 or 1]

0 <= i <= n-1, 1 <= k <= K

n 为天数，大 K 为最多交易数

此问题共 n × K × 2 种状态，全部穷举就能搞定。

for 0 <= i < n:

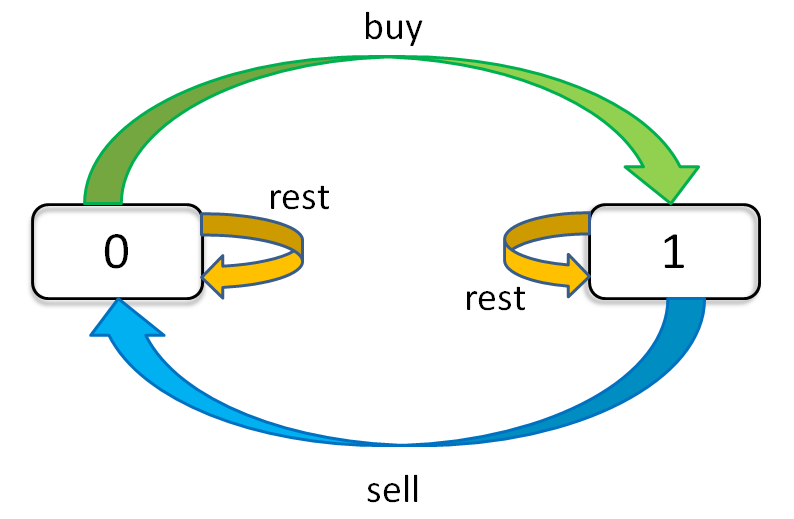
for 1 <= k <= K:

for s in {0, 1}:

dp[i][k][s] = max(buy, sell, rest)

而且我们可以用自然语言描述出每一个状态的含义，比如说dp[3][2][1]的含义就是：今天是第三天，现在手上持有股票，至多最多进行2次交易。再比如dp[2][3][0]的含义：今天是第二天，我现在手上没有股票，至多最多进行三次交易。所有我们想求的最终答案是dp[n - 1][K][0]，即最后一天，最多允许K次交易，最多获得多少利润。对于[状态]我们一般将其翻译成自然语言来理解。

现在，我们完成了[状态]的穷举，开始思考每种[状态]有哪些[选择]，应该如何更新[状态]。只看[持有状态]，可以画个状态转移图。



通过这个图可以很清楚地看到，每种状态(0和1)是如何转移而来的。根据这个图，我们来写一下状态转移方程：

dp[i][k][0] = max(dp[i - 1][k][0], dp[i - 1][k][1] + prices[i])

max(选择 rest, 选择 sell)

解释：今天我没有持有股票，有两种可能：

要么是我昨天就没持有，然后今天选择了rest，所以今天还是没有持有：

要么是我昨天持有股票，但是今天我sell了，所以我今天没有持有股票了。

dp[i][k][1] = max(dp[i - 1][k][1], dp[i - 1][k - 1][0] – prices[i])

max(选择 rest, 选择 buy)

解释：今天我持有着股票，有两种可能：

要么我昨天就持有着股票，然后今天选择 rest，所以我今天还持有着股票；

要么我昨天本没有持有，但今天我选择 buy，所以今天我就持有股票了。

这个解释应该很清楚了，如果 buy，就要从利润中减去 prices[i]，如果 sell，就要给利润增加 prices[i]。今天的最大利润就是这两种可能选择中较大的那个。而且注意 k 的限制

现在，我们已经完成了动态规划中最困难的一步：状态转移方程，接下来我们还需定义base case，即最简单的情况。

dp[-1][k][0] = 0

解释：因为 i 是从 0 开始的，所以 i = -1 意味着还没有开始，这时候的利润当然是 0 。

dp[-1][k][1] = -infinity

解释：还没开始的时候，是不可能持有股票的，用负无穷表示这种不可能。

dp[i][0][0] = 0

解释：因为 k 是从 1 开始的，所以 k = 0 意味着根本不允许交易，这时候利润当然是 0 。

dp[i][0][1] = -infinity

解释：不允许交易的情况下，是不可能持有股票的，用负无穷表示这种不可能。

把上面的状态转移方程总结一下：

base case：

dp[-1][k][0] = dp[i][0][0] = 0

dp[-1][k][1] = dp[i][0][1] = -infinity

状态转移方程：

dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])

dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])

接下来就是将上述方程转换成实际代码。

git clone、git fetch和git pull的用法及区别

1.git clone

git clone顾名思义就是将其他仓库克隆到本地，包括被clone仓库的版本变化。举个例子，你当前目录比方说是在e:/course/中，此时若想下载远程仓库，本地无需git init,直接git clone url（url是你远程仓库的地址，直接复制就可以了）。执行git clone等待clone结束，e:/course/目录下自动会有一个.git的隐藏文件夹（如果看不见，请尝试设置隐藏文件夹可见），因为是clone来的，所以.git文件夹里存放着与远程仓库一模一样的版本库记录。clone操作是一个从无到有的克隆操作，再次强调不需要git init初始化。

2.git pull

git pull是拉取远程分支更新到本地仓库的操作。比如远程仓库里的学习资料有了新内容，需要把新内容下载下来的时候，就可以用git pull命令。事实上，git pull是相当于从远程仓库获取最新版本，然后再与本地分支merge(合并)。

git pull = git fetch + git merge。注：git fetch不会进行合并，执行后需要手动执行git merge合并，而git pull拉取远程分支后直接与本地分支进行合并。更准确地说，git pull是使用给定的参数运行git fetch，并调用git merge将检索到的分支头合并到当前的分支中。

例:git pull origin master:branchtest # 将远程主机origin的master分支拉取过来，与本地的branchtest分支合并。

如果将冒号和后面的branchtest去掉，则表示将远程origin仓库的master分支拉取下来与本地当前分支合并。  
以上的git pull操作如果用git fetch来表示：

$ git fetch origin master:brantest

$ git merge brantest

相比起来，git fetch更安全也更符合实际要求，因为可以在merge前，我们可以查看更新情况，根据实际情况再决定是否合并。

git fetch更新远程代码到本地仓库

理解fetch的关键，是理解FETCH\_HEAD，FETCH\_HEAD指的是：某个branch在服务器上的最新状态。这个列表保存在.Git/FETCH\_HEAD文件中，其中每一行对应与远程服务器的一个分支。

当前分支指向的FETCH\_HEAD，就是文件第一行对应于远程服务器的一个分支。

一般来说，存在两种情况：

如果没有显式的指定远程分支, 则远程分支的master将作为默认的FETCH\_HEAD

如果指定了远程分支, 就将这个远程分支作为FETCH\_HEAD

git fetch更新本地仓库的两种方法：

方法一

$ git fetch origin master #从远程的origin仓库的master分支下载代码到本地的origin master

$ git log -p master.. origin/master #比较本地的仓库和远程参考的区别

$ git merge origin/master #把远程下载下来的代码合并到本地仓库，远程的和本地的合并

# 方法二

$ git fetch origin master:temp #从远程的origin仓库的master分支下载到本地并新建一个分支temp

$ git diff temp #比较master分支和temp分支的不同

$ git merge temp #合并temp分支到master分支

$ git branch -d temp #删除temp

git本地分支和远程分支的关联问题

1.其实在从远程分支分出来的分支都是跟踪分支（track）,当对该分支进行push和pull时，如果该分支和远程分支同名git会知道推送到远程哪个分支，从哪个远程分支同步到本地分支。其实每次克隆一个仓库时，本地新建一个master分支来track远程origin/master。如果不同名，我们需要人为指定git push origin branch\_name

2.如果本地新建了一个分支branch\_name，但是在远程没有，这时候push和pull指令就无法确定该跟踪谁,一般来说我们都会使其跟踪远程同名分支，所以可以利用git push --set-upstream origin branch\_name，这样就可以自动在远程创建一个branch\_name分支，然后本地分支会track该分支。后面再对该分支使用push和pull就自动同步。无需再指定分支。

3.跟踪远程分支

1）如果远程新建了一个分支，本地没有该分支，可以用git checkout --track origin/branch\_name，这时候本地会新建一个分支名叫branch\_name，会自动跟踪远程的同名分支branch\_name。

2）用上面中方法，得到的分支名永远和远程的分支名一样，如果想新建一个本地分支不同名字，同时跟踪一个远程分支可以利用。

git checkout -b new\_branch\_name branch\_name，这条指令本来是根据一个branch\_name分支分出一个本地分支new\_branch\_name，但是如果所根据的分支branch\_name是一个远程分支名，那么本地的分支会自动的track远程分支。建议跟踪分支和被跟踪远程分支同名。

总结：一般我们就用git push --set-upstream origin branch\_name来在远程创建一个与本地branch\_name同名的分支并跟踪；利用git checkout --track origin/branch\_name来在本地创建一个与branch\_name同名分支跟踪远程分支。

c++中STL的hash\_map

hash\_map基于hash table(哈希表)。哈希表最大的优点，就是把数据的存储和查找消耗的时间大大降低，几乎可以看成是常数时间;而代价仅仅是消耗比较多的内存。在可利用内存越来越多的情况下，用空间换时间的做法是值得的

其基本原理是：使用一个下标范围比较大的数组来存储元素。可以设计一个函数(哈希函数，也叫散列函数)，使得每个元素的关键字都与一个函数值(即数组下标,hash值)相对应，于是用这个数组单元来存储元素；也可以简单的理解为，按照关键字为每一个元素“分类”，然后将这个元素存储在相应“类”所对应的地方，称为桶。

但是不能保证每个元素的关键字与函数值是一一对应的，因此极有可能出现对于不同的元素，却计算出了相同的函数值，这样就产生了“冲突”，换句话说，就是把不同的元素分在了相同的“类”之中。 总的来说，“直接定址”与“解决冲突”是哈希表的两大特点。

hash\_map，首先分配一大片内存，形成许多桶。是利用hash函数，对key进行映射到不同区域（桶）进行保存。其插入过程是：

1.得到key

2.通过hash函数得到hash值

3.得到桶号(一般都为hash值对桶数求模)

4.存放key和value在桶内。

其取值过程是:

1.得到key

2.通过hash函数得到hash值

3.得到桶号(一般都为hash值对桶数求模)

4.比较桶的内部元素是否与key相等，若都不相等，则没有找到。

取出相等的记录的value。

hash\_map中直接地址用hash函数生成，解决冲突用比较函数解决。这里可以看出，如果每个桶内部只用一个元素，那么查找的时候只有一次比较。当许多桶内没有值时，许多查询就会更快了(指查不到的时候)。

由此可见，要实现哈希表，和用户相关的是：hash函数和比较函数。这两个参数刚好是我们在使用hash\_map时需要指定的参数。

hash\_map的用法：

先看一个简单的例子：随机给你一个ID号和ID号相应的信息，ID号的范围是1～2的31次方。如何快速保存查找。

#include <hash\_map>

#include <string>

using namespace std;

int main(){

hash\_map<int, string> mymap;

mymap[9527] = “唐伯虎点秋香”;

mymap[1000000] = “百万富翁的生活”;

mymap[10000] = “白领的工资底线”;

…

if(mymap.find(10000) != mymap.end()){

…

}

会发现和map的使用方法一样。这时你或许会问？hash函数和比较函数呢？不是要指定么？你说对了，但是在你没有指定hash函数和比较函数的时候，你会有一个缺省的函数，看看hash\_map的声明，你会更加明白。下面是SGI STL的声明：

template <class \_Key, class \_Tp, class \_HashFcn = hash<\_Key>,

class \_EqualKey = equal\_to<\_Key>,

class \_Alloc = \_\_STL\_DEFAULT\_ALLOCATOR(\_Tp) >

class hash\_map

{

...

}

也就是说，在上例中，有以下等同关系：

…

hash\_map<int, string> mymap;

//等同于

hash\_map<int, string, hash<int>, equal\_to<int>> mymap;

2.2 hash\_map的hash函数

hash<int>源码：

struct hash<int> {

size\_t operator()(int \_x) const { return \_x; }

}

这是个函数对象。在SGI STL中，提供了以下hash函数：

struct hash<char\*>

struct hash<const char\*>

struct hash<char>

struct hash<unsigned char>

struct hash<signed char>

struct hash<short>

struct hash<unsigned short>

struct hash<int>

struct hash<unsigned int>

struct hash<long>

struct hash<unsigned long>

也就是说，如果你的key使用的是以上类型中的一种，你都可以使用缺省的hash函数。当然你自己也可以定义自己的hash函数。对于自定义变量，你只能如此，例如对于string，就必须自定义hash函数。

Struct hash\_string{

size\_t operator()(const string& str) const{

unsigned long \_h = 0;

for (size\_t i = 0; i < str.size(); i++)

\_h = 5\*\_h + str[i];

return size\_t(\_h);

}

};

//如果希望利用系统定义的字符串hash函数，可以这样写：

struct hash\_string{

size\_t operator()(const string& str) const

{

return \_stl\_hash\_string(str.c\_str());

}

};

在声明自己的哈希函数时要注意以下几点：

1.使用struct，然后重载operator();

2.返回是size\_t;

3.参数是你要hash的key的类型;

4.函数是const类型的

2.3 hash\_map 的比较函数

在map中的比较函数，需要提供less函数。如果没有提供，缺省的也是less<Key>。在hash\_map中，要比较桶内的数据与key是否相等，因此需要的是是否等于的函数:equal\_to< Key>。先看看equal\_to的源码：

//本代码可以从SGI STL

//先看看binary\_function 函数声明，其实只是定义一些类型而已。

template <class \_Arg1, class \_Arg2, class \_Result>

struct binary\_function {

typedef \_Arg1 first\_argument\_type;

typedef \_Arg2 second\_argument\_type;

typedef \_Result result\_type;

};

//看看equal\_to的定义：

template <class \_Tp>

struct equal\_to : public binary\_function<\_Tp,\_Tp,bool>

{

bool operator()(const \_Tp& \_\_x, const \_Tp& \_\_y) const { return \_\_x == \_\_y; }

};

如果你使用一个自定义的数据类型，如struct mystruct，或者const char\*的字符串，如何使用比较函数？

两种方法，第一种是：重载==操作符，利用equal\_to;看看下面的例子：

struct mystruct{

int iID;

int len;

bool operator==(const mystruct & my) const{

return (iID==my.iID) && (len==my.len) ;

}

};

这样，就可以使用equal\_to<mystruct>作为比较函数了。

另一种方法是:使用函数对象。自定义一个比较函数体：

struct compare\_str{

bool operator()(const char\* pl, const char\*p2) const{

return strcmp(p1, p2)==0;

}

};

有了compare\_str，就可以使用hash\_map了。

typedef hash\_map<const char\*, string, hash<const char\*>, compare\_str> StrIntMap;

StrIntMap namemap;

namemap[“岳不群”] = “华山派掌门人，人称君子剑”;

namemap[“张三丰”] = “武当掌门人，太极拳创始人”;

namemap[“东方不败”] = “第一高手，葵花宝典”;