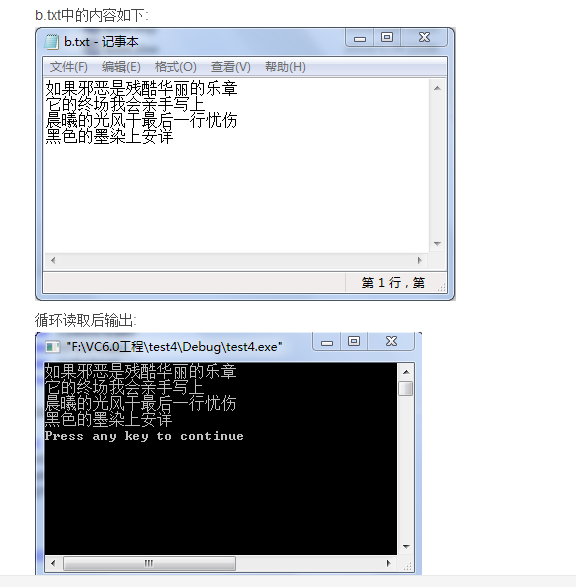
C++常用函数小结

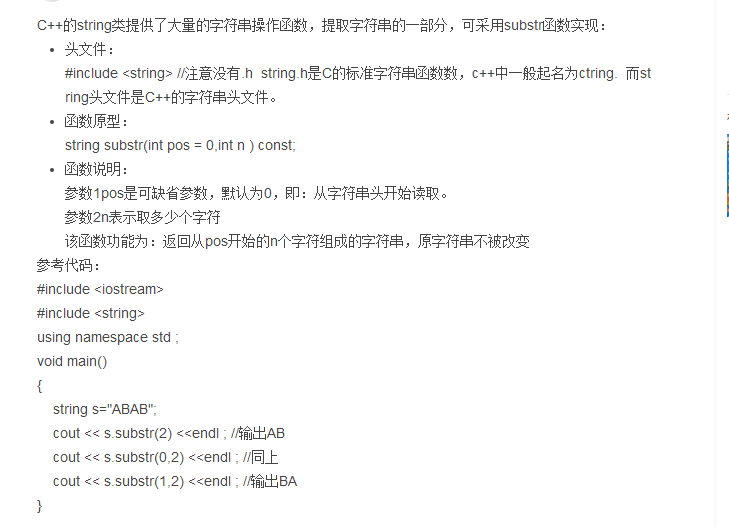
# 1.如何读取Txt文件getline

可以用getline函数，举个例子如下(从b.txt中按行循环读取内容):

#include <iostream>  
#include <fstream>  
#include <string>  
using namespace std;  
  
int main()  
{  
 int cnt=0;  
 string str;  
 ifstream fin("b.txt", ios::in);  
 if(!fin){  
 printf("The file is not exist!");  
 return -1;  
 }  
 while(getline(fin, str))  
 {  
 cnt++;  
 cout<<str<<endl;  
 }  
 return 0;  
}



# 2.如何提取文件中的字符串



cin.get()

 (2011-07-19 11:37:25)

[IMG_256](http://blog.sina.com.cn/s/javascript:;)转载▼

|  |  |
| --- | --- |
| 标签：  [杂谈](http://search.sina.com.cn/?c=blog&q=%D4%D3%CC%B8&by=tag) |  |

用法1： cin.get(字符变量名)可以用来接收字符

#include <iostream>  
using namespace std;  
main ()  
{  
char ch;  
ch=cin.get();               //或者cin.get(ch);  
cout<<ch<<endl;  
}

输入：jljkljkl  
输出：j

用法2：cin.get(字符数组名,接收字符数目)用来接收一行字符串,可以接收空格

#include <iostream>  
using namespace std;  
main ()  
{  
char a[20];  
cin.get(a,20);  
cout<<a<<endl;  
}

输入：jkl jkl jkl  
输出：jkl jkl jkl

输入：abcdeabcdeabcdeabcdeabcde （输入25个字符）  
输出：abcdeabcdeabcdeabcd              （接收19个字符+1个'\0'）

用法3：cin.get(无参数)没有参数主要是用于舍弃输入流中的不需要的字符,或者舍弃回车,弥补cin.get(字符数组名,接收字符数目)的不足.

这个我还不知道怎么用，知道的前辈请赐教；

3、cin.getline()   // 接受一个字符串，可以接收空格并输出

#include <iostream>  
using namespace std;  
main ()  
{  
char m[20];  
cin.getline(m,5);  
cout<<m<<endl;  
}

输入：jkljkljkl  
输出：jklj

接受5个字符到m中，其中最后一个为'\0'，所以只看到4个字符输出；

如果把5改成20：  
输入：jkljkljkl  
输出：jkljkljkl

输入：jklf fjlsjf fjsdklf  
输出：jklf fjlsjf fjsdklf

//延伸：  
//cin.getline()实际上有三个参数，cin.getline(接受字符串的看哦那间m,接受个数5,结束字符)  
//当第三个参数省略时，系统默认为'\0'  
//如果将例子中cin.getline()改为cin.getline(m,5,'a');当输入jlkjkljkl时输出jklj，输入jkaljkljkl时，输出jk

当用在多维数组中的时候，也可以用cin.getline(m[i],20)之类的用法：

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

main ()  
{  
char m[3][20];  
for(int i=0;i<3;i++)  
{  
cout<<"\n请输入第"<<i+1<<"个字符串："<<endl;  
cin.getline(m[i],20);  
}

cout<<endl;  
for(int j=0;j<3;j++)  
cout<<"输出m["<<j<<"]的值:"<<m[j]<<endl;

}

# 3.C++中getline函数与cin.getline函数的区别

1、getline(infile , s , '\n') , s只能是 string s

infile为读取流对象，ifstream infile;

S为字符串变量名。 string s;

即将infile读取的一行字符串保存到字符串变量s中

getline属于string类的字符读取函数，这个函数的第三个参数可以不写(getling(cin , s))，第三个参数默认是'\0'，这种情况下如果要结束一个字符串输入，对于有些编译器（VC , VS）， 必须得连续输入两个换行。

对于getline(cin , s , 'z') ，能识别空格和换行

对于getline只要截止字符（第三个参数）设置得当，能读取多行

cin.getline(s , 100 , '/n') ， 只能是 char s[];

s为字符数组或字符指针，char s[];

100为读取字符的数目，大于0任意，则定义s[m]时s的元素个数m要比读取的字符数组大。‘/n’为终止标志字符，可任意

cin.getline属于ostream ， 和getline类似，只是只能对char s[]类型进行读取，除这点外，其他和getline都相同。

# 4. year=atoi(s.substr(0,4).c\_str());

或者

s1=s.substr(0,4);

Year=atoi(s1.c\_str());

其实这里面是有三个函数的,分别是atoi(),substr(),c\_str().先说说第一个函数atoi()吧.  
  
 atoi()函数原型为: int atoi(char \*str),用途是将字符串转换成一个整数值,str是待转化成整数值的字符串.成功责返回转化后的整数值,失败返回0.  
 substr()函数原型为:basic string::substr(string,start,length),也可把string移到外面,为string &a,a.substr(start,length),其中a是待截取的字符串,start表示从截取开始的前一位,length表示截取长度,例如string &a="hello world",则a.substr(6,5)=world.  
 c\_str()函数原型为:const char \*c\_str(),如果要将string对象，转化为char\*对象，c\_str()提供了这样一种方法，它返回一个客户程序可读不可改的指向字符数组的指针。  
  
所以year=atoi(s.substr(0,4).c\_str())的作用就是,截取string型的对象s,并转化为char\*对象,然后将此字符串转换成一个整数值,赋值给year(year是int型).

2. string/array to int/float

C/C++语言提供了几个标准库函数，可以将字符串转换为任意类型(整型、长整型、浮点型等)。

● atof()：将字符串转换为双精度浮点型值。

● atoi()：将字符串转换为整型值。

● atol()：将字符串转换为长整型值。

● strtod()：将字符串转换为双精度浮点型值，并报告不能被转换的所有剩余数字。

● strtol()：将字符串转换为长整值，并报告不能被转换的所有剩余数字。

● strtoul()：将字符串转换为无符号长整型值，并报告不能被转换的所有剩余数字。

**下面是整型转字符串**

**int number = 123456;**

**char a[25];**

**\_itoa\_s(number, a, 10); //按十进制转换**

**string str = (string)a;**

**cout << str << endl;**

3、 atol——将字符串转换成长整型数

头文件：stdlib.h

语法：long atol(const char \*nptr)

4、 \_ecvt——将浮点数转换成字符串

头文件：stdlib.h

语法：char \*\_ecvt( double value, int count, int \*dec, int \*sign );

5、 \_fcvt——将浮点数转换成字符串

头文件：stdlib.h

语法：char \*\_fcvt( double value, int count, int \*dec, int \*sign )

6、 \_itoa——将整形数转换成字符串

头文件：stdlib.h

语法：char \*\_itoa( int value, char \*string, int radix )

7、 \_ltoa——将长整形数转换成字符串

头文件：stdlib.h

语法：char \*\_ltoa( long value, char \*string, int radix )

8、 Strtod——将字符串转换成双精度数

头文件：stdlib.h

语法：double strtod(const char \*nptr,char \*\*endptr)

9、 Strtol——将字符串转换成长整型数

头文件：stdlib.h

语法：long int strtol(const char \*nptr,char \*\*endptr,int base)

10、 Strtoul——将字符串转换成无符号长整型数

头文件：stdlib.h

语法：unsigned long int strtoul(const char \*nptr, char \*\*endptr,int base)

11、 \_toascii——将整型数转换成合法的ASCII 码字符

头文件：ctype.h

语法：int toascii(int c)

12、 \_ultoa——转换一个无符号长整型数为字符串

头文件：stdlib.h

语法：char \*ultoa(unsigned long value, char \*string, int radix)

说明:ultoa函数把 value转换成一个以空格结尾的字符串,并存储在string中(至多33个字节)，不执行上溢出检查。radix指出value的基数,radi 必须在2-36的范围内。

程序例:

#include

#include

int main( void )

{

unsigned long lnumber = 3123456789L;

char string[25];

ultoa(lnumber,string,10);

printf("string = %s unsigned long = %lu\n",string,lnumber);

return 0;

}

13、 toupper——将小写字母转换成大写字母

头文件：ctype.h

语法：int toupper(int c)

14、 tolower——将大写字母转换成小写字母

头文件：stdlib.h

语法：int tolower(int c)

# 5.C++中的取整函数

C语言有以下几种取整方法：

1、直接赋值给整数变量。如：

int i = 2.5; 或 i = (int) 2.5;

这种方法采用的是舍去小数部分

2、C/C++中的整数除法运算符“/”本身就有取整功能(int / int)，但是整数除法对负数的取整结果和使用的C编译器有关。

3、使用floor函数。floor(x)返回的是小于或等于x的最大整数。如：

floor(2.5) = 2

floor(-2.5) = -3

4、使用ceil函数。ceil(x)返回的是大于x的最小整数。如：

ceil(2.5) = 3

ceil(-2.5) = -2

floor()是向负无穷大舍入，floor(-2.5) = -3；ceil()是向正无穷大舍入，ceil(-2.5) = -2。

# 6.math.h 数学函数库

一些数学计算的公式的具体实现是放在math.h里,具体有：

1 三角函数

double sin (double);

double cos (double);

double tan (double);

2 反三角函数

double asin (double); 结果介于[-PI/2, PI/2]

double acos (double); 结果介于[0, PI]

double atan (double); 反正切(主值), 结果介于[-PI/2, PI/2]

double atan2 (double, double); 反正切(整圆值), 结果介于[-PI, PI]

3 双曲三角函数

double sinh (double);

double cosh (double);

double tanh (double);

4 指数与对数

double exp (double);

double pow (double, double);

double sqrt (double);

double log (double); 以e为底的对数

double log10 (double);c++中自然对数函数：log(N) 以10为底：log10(N)但没有以2为底的函数但是可以用换底公式解 决：log2(N)=log10(N)/log10(2)

5 取整

double ceil (double); 取上整

double floor (double); 取下整

6 绝对值

double fabs (double);

7 标准化浮点数

double frexp (double f, int \*p); 标准化浮点数, f = x \* 2^p, 已知f求x, p ( x介于[0.5, 1] )

double ldexp (double x, int p); 与frexp相反, 已知x, p求f

8 取整与取余

double modf (double, double\*); 将参数的整数部分通过指针回传, 返回小数部分

double fmod (double, double); 返回两参数相除的余数

source: 《C & C++ Code Capsules》

# 7.C/C++头文件一览

C

#include <assert.h>　　　　//设定插入点

#include <ctype.h>　　　　 //字符处理

#include <errno.h>　　　　 //定义错误码

#include <float.h>　　　　 //浮点数处理

#include <iso646.h> //对应各种运算符的宏

#include <limits.h>　　　　//定义各种数据类型最值的常量

#include <locale.h>　　　　//定义本地化C函数

#include <math.h>　　　　　//定义数学函数

#include <setjmp.h> //异常处理支持

#include <signal.h> //信号机制支持

#include <stdarg.h> //不定参数列表支持

#include <stddef.h> //常用常量

#include <stdio.h>　　　　 //定义输入／输出函数

#include <stdlib.h>　　　　//定义杂项函数及内存分配函数

#include <string.h>　　　　//字符串处理

#include <time.h>　　　　　//定义关于时间的函数

#include <wchar.h>　　　　 //宽字符处理及输入／输出

#include <wctype.h>　　　　//宽字符分类

传统C++

#include <fstream.h>　　　 //改用<fstream>

#include <iomanip.h>　　　 //改用<iomainip>

#include <iostream.h>　　　//改用<iostream>

#include <strstrea.h>　　　//该类不再支持，改用<sstream>中的stringstream

————————————————————————————————

## 标准C++

#include <algorithm>　　　 //STL 通用算法

#include <bitset>　　　　　//STL 位集容器

#include <cctype> //字符处理

#include <cerrno> 　　　　 //定义错误码

#include <cfloat>　　　　 //浮点数处理

#include <ciso646> //对应各种运算符的宏

#include <climits> 　　　　//定义各种数据类型最值的常量

#include <clocale> 　　　　//定义本地化函数

#include <cmath> 　　　　　//定义数学函数

#include <complex>　　　　 //复数类

#include <csignal> //信号机制支持

#include <csetjmp> //异常处理支持

#include <cstdarg> //不定参数列表支持

#include <cstddef> //常用常量

#include <cstdio> 　　　　 //定义输入／输出函数

#include <cstdlib> 　　　　//定义杂项函数及内存分配函数

#include <cstring> 　　　　//字符串处理

#include <ctime> 　　　　　//定义关于时间的函数

#include <cwchar> 　　　　 //宽字符处理及输入／输出

#include <cwctype> 　　　　//宽字符分类

#include <deque>　　　　　 //STL 双端队列容器

#include <exception>　　　 //异常处理类

#include <fstream> 　　　 //文件输入／输出

#include <functional>　　　//STL 定义运算函数（代替运算符）

#include <limits> 　　　　 //定义各种数据类型最值常量

#include <list>　　　　　　//STL 线性列表容器

#include <locale> //本地化特定信息

#include <map>　　　　　　 //STL 映射容器

#include <memory> //STL通过分配器进行的内存分配

#include<new> //动态内存分配

#include <numeric> //STL常用的数字操作

#include <iomanip> 　　　 //参数化输入／输出

#include <iOS>　　　　　　 //基本输入／输出支持

#include <iosfwd>　　　　　//输入／输出系统使用的前置声明

#include <iostream> 　　　//数据流输入／输出

#include <istream>　　　　 //基本输入流

#include <iterator> //STL迭代器

#include <ostream>　　　　 //基本输出流

#include <queue>　　　　　 //STL 队列容器

#include <set>　　　　　　 //STL 集合容器

#include <sstream>　　　　 //基于字符串的流

#include <stack>　　　　　 //STL 堆栈容器

#include <stdexcept>　　　 //标准异常类

#include <streambuf>　　　 //底层输入／输出支持

#include <string>　　　　　//字符串类

#include <typeinfo> //运行期间类型信息

#include <utility>　　　　 //STL 通用模板类

#include <valarray> //对包含值的数组的操作

#include <vector>　　　　　//STL 动态数组容器

————————————————————————————————

C99增加的部分

#include <complex.h>　　 //复数处理

#include <fenv.h>　　　　//浮点环境

#include <inttypes.h>　　//整数格式转换

#include <stdbool.h>　　 //布尔环境

#include <stdint.h>　　　//整型环境

#include <tgmath.h>　　　//通用类型数学宏

头文件 ctype.h

字符处理函数: 本类别函数用于对单个字符进行处理，包括字符的类别测试和字符的大小写转换

----------------------------------------

字符测试是否字母和数字 isalnum

是否字母 isalpha

是否控制字符 iscntrl

是否数字 isdigit

是否可显示字符(除空格外) isgraph

是否可显示字符(包括空格) isprint

是否既不是空格，又不是字母和数字的可显示字符 ispunct

是否空格 isspace

是否大写字母 isupper

是否16进制数字(0－9，A-F)字符 isxdigit

字符大小写转换函数 转换为大写字母 toupper

转换为小写字母 tolower

头文件 local.h

地区化: 本类别的函数用于处理不同国家的语言差异。

----------------------------------------

地区控制 地区设置 setlocale

数字格式约定查询 国家的货币、日期、时间等的格式转换 localeconv

头文件 math.h

数学函数: 本分类给出了各种数学计算函数，必须提醒的是ANSIC标准中的数据格式并不符合IEEE754标准，一些C语言编译器却遵循IEEE754(例如frinklin C51)

----------------------------------------

反余弦 acos

反正弦 asin

反正切 atan

反正切2 atan2

余弦 cos

正弦 sin

正切 tan

双曲余弦 cosh

双曲正弦 sinh

双曲正切 tanh

指数函数 exp

指数分解函数 frexp

乘积指数函数 fdexp

自然对数 log

以10为底的对数 log10

浮点数分解函数 modf

幂函数 pow

平方根函数 sqrt

求下限接近整数 ceil

绝对值 fabs

求上限接近整数 floor

求余数 fmod

头文件 setjmp.h io.h

本分类函数用于实现在不同底函数之间直接跳转代码。

----------------------------------------

保存调用环境 setjmp

恢复调用环境 longjmp

头文件 signal.h

信号处理: 该分类函数用于处理那些在程序执行过程中发生例外的情况。

----------------------------------------

指定信号处理函数 signal

发送信号 raise

头文件 stdarg.h

可变参数处理: 本类函数用于实现诸如printf,scanf等参数数量可变底函数。

----------------------------------------

可变参数访问宏

可变参数开始宏 va\_start

可变参数结束宏 va\_end

可变参数访问宏 访问下一个可变参数宏 va\_arg

头文件 stdio.h

输入输出函数:该分类用于处理包括文件、控制台等各种输入输出设备，各种函数以“流”的方式实现

----------------------------------------

删除文件 remove

修改文件名称 rename

生成临时文件名称 tmpfile

得到临时文件路径 tmpnam

文件访问 关闭文件 fclose

刷新缓冲区 fflush

打开文件 fopen

将已存在的流指针和新文件连接 freopen

设置磁盘缓冲区 setbuf

设置磁盘缓冲区 setvbuf

格式化输入与输出函数

格式输出 fprintf

格式输入 fscanf

格式输出(控制台) printf

格式输入(控制台) scanf

格式输出到缓冲区 sprintf

从缓冲区中按格式输入 sscanf

格式化输出 vfprintf

格式化输出 vprintf

格式化输出 vsprintf

字符输入输出函数

输入一个字符 fgetc

字符串输入 fgets

字符输出 fputc

字符串输出 fputs

字符输入(控制台) getc

字符输入(控制台) getchar

字符串输入(控制台) gets

字符输出(控制台) putc

字符输出(控制台) putchar

字符串输出(控制台) puts

字符输出到流的头部 ungetc

直接输入输出

直接流读操作 fread

直接流写操作 fwrite

文件定位函数

得到文件位置 fgetpos

文件位置移动 fseek

文件位置设置 fsetpos

得到文件位置 ftell

文件位置复零位 remind

错误处理函数

错误清除 clearerr

文件结尾判断 feof

文件错误检测 ferror

得到错误提示字符串 perror

头文件 stdlib.h

实用工具函数: 本分类给出了一些函数无法按以上分类，但又是编程所必须要的。

----------------------------------------

字符串转换函数

字符串转换为整数 atoi

字符串转换为长整数 atol

字符串转换为浮点数 strtod

字符串转换为长整数 strtol

字符串转换为无符号长整型 strtoul

伪随机序列产生函数

产生随机数 rand

设置随机函数的起动数值 srand

存储管理函数

分配存储器 calloc

释放存储器 free

存储器分配 malloc

重新分配存储器 realloc

环境通信

中止程序 abort

退出程序执行，并清除环境变量 atexit

退出程序执行 exit

读取环境参数 getenv

程序挂起，临时执行一个其他程序 system

搜索和排序工具 二分查找(数据必须已排序) bsearch

快速排序 qsort

整数运算函数 求绝对值 abs

得到除法运算底商和余数 div

求长整形底绝对值 labs

求长整形除法的商和余数 ldiv

多字节字符函数 得到多字节字符的字节数 mblen

得到多字节字符的字节数 mbtowc

多字节字符转换 wctomb

多字节字符的字符串操作 将多字节串转换为整数数组 mbstowcs

将多字节串转换为字符数组 mcstowbs

## 头文件 string.h

学习网址：http://www.cnblogs.com/rosesmall/category/364442.html

字符串处理: 本分类的函数用于对字符串进行合并、比较等操作

----------------------------------------

字符串拷贝 块拷贝(目的和源存储区不可重叠) memcpy

块拷贝(目的和源存储区可重叠) memmove

串拷贝 strcpy

按长度的串拷贝 strncpy

字符串连接函数 串连接 strcat

按长度连接字符串 strncat

串比较函数 块比较 memcmp

字符串比较 strcmp

字符串比较(用于非英文字符) strcoll

按长度对字符串比较 strncmp

字符串转换 strxfrm

字符与字符串查找 字符查找 memchr

字符查找 strchr

字符串查找 strcspn

字符串查找 strpbrk

字符串查找 strspn

字符串查找 strstr

if((strstr(dat,"ORB:CMB CLK:CMB") != NULL) || (strstr(dat,"IIR 0.000") != NULL))

//查找char dat[]字符数组中是否存在”ORB:CMB CLK:CMB"以及"IIR 0.000"，若不存在，NULL.

//该判断语句作业：若检测字符串中存在目标字符串，则执行if条件语句。

字符串分解 strtok

杂类函数 字符串设置 memset

错误字符串映射 strerror

求字符串长度 strlen

### 下面为string.h文件中函数的详细用法，附加实例：

1、strcpy

　　函数名: strcpy

　　功 能: 拷贝一个字符串到另一个

　　用 法: char \*strcpy(char \*destin, char \*source);

　　程序例:

　　#include <stdio.h>

　　#include <string.h>

　　int main(void)

　　{

　　char string[10];

　　char \*str1 = "abcdefghi";

　　strcpy(string, str1);

printf("%s\n", string);

　　return 0;

　　}

2、strcat

　　函数名: strcat

　　功 能: 字符串拼接函数

　　用 法: char \*strcat(char \*destin, char \*source);

　　程序例:

　　#include <string.h>

　　#include <stdio.h>

　　int main(void)

　　{

　　char destination[25];

　　char \*blank = " ", \*c = "C++", \*Borland = "Borland";

　　strcpy(destination, Borland);

　　strcat(destination, blank);

　　strcat(destination, c);

　　printf("%s\n", destination);

　　return 0;

　　}

3、strchr

　　函数名: strchr

　　功 能: 在一个串中查找给定字符的第一个匹配之处\

　　用 法: char \*strchr(char \*str, char c);

　　程序例:

　　#include <string.h>

　　#include <stdio.h>

　　int main(void)

　　{

　　char string[15];

　　char \*ptr, c = 'r';

　　strcpy(string, "This is a string");

　　ptr = strchr(string, c);

　　if (ptr)

　　printf("The character %c is at position: %d\n", c, ptr-string);

　　else

　　printf("The character was not found\n");

　　return 0;

　　}

4、strcmp

　　函数名: strcmp

　　功 能: 串比较

　　用 法: int strcmp(char \*str1, char \*str2);

　　看Asic码，str1>str2，返回值 > 0；两串相等，返回0

　　程序例:

　　#include <string.h>

　　#include <stdio.h>

　　int main(void)

　　{

　　char \*buf1 = "aaa", \*buf2 = "bbb", \*buf3 = "ccc";

　　int ptr;

　　ptr = strcmp(buf2, buf1);

　　if (ptr > 0)

　　printf("buffer 2 is greater than buffer 1\n");

　　else

　　printf("buffer 2 is less than buffer 1\n");

　　ptr = strcmp(buf2, buf3);

　　if (ptr > 0)

　　printf("buffer 2 is greater than buffer 3\n");

　　else

　　printf("buffer 2 is less than buffer 3\n");

　　return 0;

　　}

5、strncmpi

　　函数名: strncmpi

　　功 能: 将一个串中的一部分与另一个串比较, 不管大小写

　　用 法: int strncmpi(char \*str1, char \*str2, unsigned maxlen);

　　程序例:

　　#include <string.h>

　　#include <stdio.h>

　　int main(void)

　　{

　　char \*buf1 = "BBB", \*buf2 = "bbb";

　　int ptr;

　　ptr = strcmpi(buf2, buf1);

　　if (ptr > 0)

　　printf("buffer 2 is greater than buffer 1\n");

　　if (ptr < 0)

　　printf("buffer 2 is less than buffer 1\n");

　　if (ptr == 0)

　　printf("buffer 2 equals buffer 1\n");

　　return 0;

　　}

6、strcpy

　　函数名: strcpy

　　功 能: 串拷贝

　　用 法: char \*strcpy(char \*str1, char \*str2);

　　程序例:

　　#include <stdio.h>

　　#include <string.h>

　　int main(void)

　　{

　　char string[10];

　　char \*str1 = "abcdefghi";

　　strcpy(string, str1);

　　printf("%s\n", string);

　　return 0;

　　}

7、strcspn

　　函数名: strcspn

　　功 能: 在串中查找第一个给定字符集内容的段

　　用 法: int strcspn(char \*str1, char \*str2);

　　程序例:

　　#include <stdio.h>

　　#include <string.h>

　　#include <alloc.h>

　　int main(void)

　　{

　　char \*string1 = "1234567890";

　　char \*string2 = "747DC8";

　　int length;

　　length = strcspn(string1, string2);

　　printf("Character where strings intersect is at position %d\n", length);

　　return 0;

　　}

8、strdup

　　函数名: strdup

　　功 能: 将串拷贝到新建的位置处

　　用 法: char \*strdup(char \*str);

　　程序例:

　　#include <stdio.h>

　　#include <string.h>

　　#include <alloc.h>

　　int main(void)

　　{

　　char \*dup\_str, \*string = "abcde";

　　dup\_str = strdup(string);

　　printf("%s\n", dup\_str);

　　free(dup\_str);

　　return 0;

　　}

9、stricmp

　　函数名: stricmp

　　功 能: 以大小写不敏感方式比较两个串

　　用 法: int stricmp(char \*str1, char \*str2);

　　程序例:

　　#include <string.h>

　　#include <stdio.h>

　　int main(void)

　　{

　　char \*buf1 = "BBB", \*buf2 = "bbb";

　　int ptr;

　　ptr = stricmp(buf2, buf1);

　　if (ptr > 0)

　　printf("buffer 2 is greater than buffer 1\n");

　　if (ptr < 0)

　　printf("buffer 2 is less than buffer 1\n");

　　if (ptr == 0)

　　printf("buffer 2 equals buffer 1\n");

　　return 0;

　　}

10、strerror

　　函数名: strerror

　　功 能: 返回指向错误信息字符串的指针

　　用 法: char \*strerror(int errnum);

　　程序例:

　　#include <stdio.h>

　　#include <errno.h>

　　int main(void)

　　{

　　char \*buffer;

　　buffer = strerror(errno);

　　printf("Error: %s\n", buffer);

　　return 0;

　　}

11、strcmpi

　　函数名: strcmpi

　　功 能: 将一个串与另一个比较, 不管大小写

　　用 法: int strcmpi(char \*str1, char \*str2);

　　程序例:

　　#include <string.h>

　　#include <stdio.h>

　　int main(void)

　　{

　　char \*buf1 = "BBB", \*buf2 = "bbb";

　　int ptr;

　　ptr = strcmpi(buf2, buf1);

　　if (ptr > 0)

　　printf("buffer 2 is greater than buffer 1\n");

　　if (ptr < 0)

　　printf("buffer 2 is less than buffer 1\n");

　　if (ptr == 0)

　　printf("buffer 2 equals buffer 1\n");

　　return 0;

　　}

　　函数名: strncmp

　　功 能: 串比较

　　用 法: int strncmp(char \*str1, char \*str2, int maxlen);

　　程序例:

　　#include <string.h>

　　#include <stdio.h>

　　int main(void)

　　{

　　char \*buf1 = "aaabbb", \*buf2 = "bbbccc", \*buf3 = "ccc";

　　int ptr;

　　ptr = strncmp(buf2,buf1,3);

　　if (ptr > 0)

　　printf("buffer 2 is greater than buffer 1\n");

　　else

　　printf("buffer 2 is less than buffer 1\n");

　　ptr = strncmp(buf2,buf3,3);

　　if (ptr > 0)

　　printf("buffer 2 is greater than buffer 3\n");

　　else

　　printf("buffer 2 is less than buffer 3\n");

　　return(0);

　　}

12、strncpy

　　函数名: strncpy

　　功 能: 串拷贝

　　用 法: char \*strncpy(char \*destin, char \*source, int maxlen);

　　程序例:

　　#include <stdio.h>

　　#include <string.h>

　　int main(void)

　　{

　　char string[10];

　　char \*str1 = "abcdefghi";

　　strncpy(string, str1, 3);

　　string[3] = '\0';

　　printf("%s\n", string);

　　return 0;

　　}

13、strnicmp

　　函数名: strnicmp

　　功 能: 不注重大小写地比较两个串

　　用 法: int strnicmp(char \*str1, char \*str2, unsigned maxlen);

　　程序例:

　　#include <string.h>

　　#include <stdio.h>

　　int main(void)

　　{

　　char \*buf1 = "BBBccc", \*buf2 = "bbbccc";

　　int ptr;

　　ptr = strnicmp(buf2, buf1, 3);

　　if (ptr > 0)

　　printf("buffer 2 is greater than buffer 1\n");

　　if (ptr < 0)

　　printf("buffer 2 is less than buffer 1\n");

　　if (ptr == 0)

　　printf("buffer 2 equals buffer 1\n");

　　return 0;

　　}

14、strnset

　　函数名: strnset

　　功 能: 将一个字符串前n个字符都设为指定字符

　　用 法: char \*strnset(char \*str, char ch, unsigned n);

　　程序例:

　　#include <stdio.h>

　　#include <string.h>

　　int main(void)

　　{

　　char \*string = "abcdefghijklmnopqrstuvwxyz";

　　char letter = 'x';

　　printf("string before strnset: %s\n", string);

　　strnset(string, letter, 13);

　　printf("string after strnset: %s\n", string);

　　return 0;

　　}

15、strpbrk

　　函数名: strpbrk

　　功 能: 在串中查找给定字符集中的字符

　　用 法: char \*strpbrk(char \*str1, char \*str2);

　　程序例:

　　#include <stdio.h>

　　#include <string.h>

　　int main(void)

　　{

　　char \*string1 = "abcdefghijklmnopqrstuvwxyz";

　　char \*string2 = "onm";

　　char \*ptr;

　　ptr = strpbrk(string1, string2);

　　if (ptr)

　　printf("strpbrk found first character: %c\n", \*ptr);

　　else

　　printf("strpbrk didn't find character in set\n");

　　return 0;

　　}

16、strrchr

　　函数名: strrchr

　　功 能: 在串中查找指定字符的最后一个出现

　　用 法: char \*strrchr(char \*str, char c);

　　程序例:

　　#include <string.h>

　　#include <stdio.h>

　　int main(void)

　　{

　　char string[15];

　　char \*ptr, c = 'r';

　　strcpy(string, "This is a string");

　　ptr = strrchr(string, c);

　　if (ptr)

　　printf("The character %c is at position: %d\n", c, ptr-string);

　　else

　　printf("The character was not found\n");

　　return 0;

　　}

17、strrev

　　函数名: strrev

　　功 能: 串倒转

　　用 法: char \*strrev(char \*str);

　　程序例:

　　#include <string.h>

　　#include <stdio.h>

　　int main(void)

　　{

　　char \*forward = "string";

　　printf("Before strrev(): %s\n", forward);

　　strrev(forward);

　　printf("After strrev(): %s\n", forward);

　　return 0;

　　}

18、strset

　　函数名: strset

　　功 能: 将一个串中的所有字符都设为指定字符

　　用 法: char \*strset(char \*str, char c);

　　程序例:

　　#include <stdio.h>

　　#include <string.h>

　　int main(void)

　　{

　　char string[10] = "123456789";

　　char symbol = 'c';

　　printf("Before strset(): %s\n", string);

　　strset(string, symbol);

　　printf("After strset(): %s\n", string);

　　return 0;

　　}

19、strspn

　　函数名: strspn

　　功 能: 在串中查找指定字符集的子集的第一次出现

　　用 法: int strspn(char \*str1, char \*str2);

　　程序例:

　　#include <stdio.h>

　　#include <string.h>

　　#include <alloc.h>

　　int main(void)

　　{

　　char \*string1 = "1234567890";

　　char \*string2 = "123DC8";

　　int length;

　　length = strspn(string1, string2);

　　printf("Character where strings differ is at position %d\n", length);

　　return 0;

　　}

20、strstr

　　函数名: strstr

　　功 能: 在串中查找指定字符串的第一次出现

　　用 法: char \*strstr(char \*str1, char \*str2);

　　程序例:

　　#include <stdio.h>

　　#include <string.h>

　　int main(void)

　　{

　　char \*str1 = "Borland International", \*str2 = "nation", \*ptr;

　　ptr = strstr(str1, str2);

　　printf("The substring is: %s\n", ptr);

　　return 0;

　　}

21、strtod

　　函数名: strtod

　　功 能: 将字符串转换为double型值

　　用 法: double strtod(char \*str, char \*\*endptr);

　　程序例:

　　#include <stdio.h>

　　#include <stdlib.h>

　　int main(void)

　　{

　　char input[80], \*endptr;

　　double value;

　　printf("Enter a floating point number:");

　　gets(input);

　　value = strtod(input, &endptr);

　　printf("The string is %s the number is %lf\n", input, value);

　　return 0;

　　}

22、strtok

　　函数名: strtok

　　功 能: 查找由在第二个串中指定的分界符分隔开的单词

　　用 法: char \*strtok(char \*str1, char \*str2);

　　程序例:

　　#include <string.h>

　　#include <stdio.h>

　　int main(void)

　　{

　　char input[16] = "abc,d";

　　char \*p;

　　/\* strtok places a NULL terminator

　　in front of the token, if found \*/

　　p = strtok(input, ",");

　　if (p) printf("%s\n", p);

　　/\* A second call to strtok using a NULL

　　as the first parameter returns a pointer

　　to the character following the token \*/

　　p = strtok(NULL, ",");

　　if (p) printf("%s\n", p);

　　return 0;

　　}

23、strtol

　　函数名: strtol

　　功 能: 将串转换为长整数

　　用 法: long strtol(char \*str, char \*\*endptr, int base);

　　程序例:

　　#include <stdlib.h>

　　#include <stdio.h>

　　int main(void)

　　{

　　char \*string = "87654321", \*endptr;

　　long lnumber;

　　/\* strtol converts string to long integer \*/

　　lnumber = strtol(string, &endptr, 10);

　　printf("string = %s long = %ld\n", string, lnumber);

　　return 0;

　　}

24、strupr

　　函数名: strupr

　　功 能: 将串中的小写字母转换为大写字母

　　用 法: char \*strupr(char \*str);

　　程序例:

　　#include <stdio.h>

　　#include <string.h>

　　int main(void)

　　{

　　char \*string = "abcdefghijklmnopqrstuvwxyz", \*ptr;

　　/\* converts string to upper case characters \*/

　　ptr = strupr(string);

　　printf("%s\n", ptr);

　　return 0;

　　}

25、swab

　　函数名: swab

　　功 能: 交换字节

　　用 法: void swab (char \*from, char \*to, int nbytes);

　　程序例:

　　#include <stdlib.h>

　　#include <stdio.h>

　　#include <string.h>

　　char source[15] = "rFna koBlrna d";

　　char target[15];

　　int main(void)

　　{

　　swab(source, target, strlen(source));

　　printf("This is target: %s\n", target);

　　return 0;

　　原型：extern char \*strstr(char \*haystack, char \*needle);

　　\*所在头文件：#include <string.h>

　　\*功能：从字符串haystack中寻找needle第一次出现的位置（不比较结束符NULL)。

　　\*说明：返回指向第一次出现needle位置的指针，如果没找到则返回NULL。

头文件 time.h

日期和时间函数: 本类别给出时间和日期处理函数

----------------------------------------

时间操作函数得到处理器时间 clock

得到时间差 difftime

设置时间 mktime

得到时间 time

时间转换函数 得到以ASCII码表示的时间 asctime

得到字符串表示的时间 ctime

得到指定格式的时间 strftime

序号 库类别 头文件

----------------------------------------

1 错误处理 errno.h

2 字符处理 ctyphe.

3 地区化 local.h

4 数学函数 math.h

5 信号处理 signal.h

6 输入输出 stdio.h

7 实用工具程序 stdlib.h

8 字符串处理 string.h

# 删除数组中的某一元素重新排列问题

#include "stdafx.h"

#include<iostream>

using namespace std;

int \_tmain(int argc, \_TCHAR\* argv[])

{

int a[5] = { 1, 2, 3, 2, 5 };

int i, j, c(0);

for (i = 0; i < 5; i++)

if (a[i] == 2)

c++;

int b[5];

int \*p = a;

/\* for循环

for (j = 0; j < 5;p++)

{

if (\*p!=2)

{

b[j] = \*p;

j++;

}

}

\*/

/\*0\*\*do-while循环

j = 0;

do

{

if (\*p != 2)

{

b[j] = \*p;

j++;

}

p++;

} while (j < 5);

\*/

//while循环

j = 0;

while (j < 5)

{

if (\*p != 2)

{

b[j] = \*p;

j++;

}

p++;

}

for (i = 0; i < 5; i++)

{

cout << a[i] << " ";

}

cout << endl;

cout <<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" <<endl;

for (i = 0; i < 5-c; i++)

cout << b[i] << " ";

cout << endl;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<< endl;

system("pause");

return 0;

}

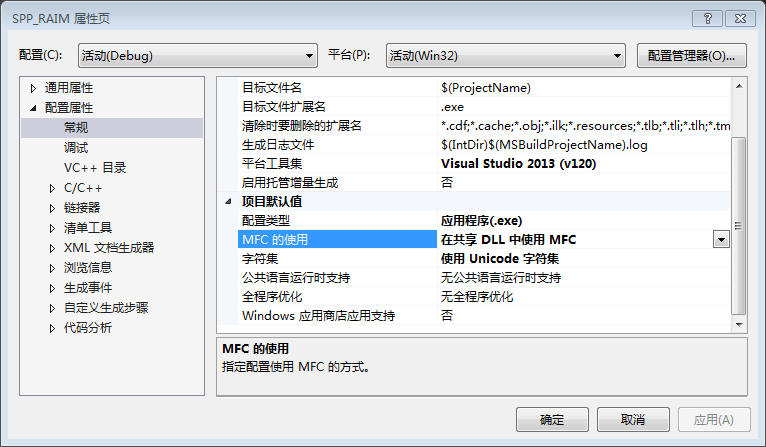
# CArray动态数组

主要是用于创建长度可扩展的的动态数组，储存数据。

template<class TYPE,class ARG\_TYPE>class CArray:public CObject

参数

使用时要设置配置属性，见附图设置！



CArray预定义大小后，使用Add函数会在末尾添加；而vector预定义大小后，使用push\_back函数仍然会在初始添加！！！

例子：

#include”afxtempl.h”

int \_tmain(int argc, \_TCHAR\* argv[])

{

CArray<int, int&>a;

int i,j;

for (j=1,i = 0; i < 10;i++,j++)

a.Add(j);

for (i = 0; i < 10; i++)

cout << a[i] << " ";

cout << endl;

int b = a.GetSize();

cout << b << endl;

a.RemoveAll();

system("pause");

return 0;

}

TYPE

模板参数指定存储在数组中的对象的类型。TYPE是一个由CArray返回的参数。ARG\_TYPE

模板参数指定用于访问存储在数组中对象的参数类型。通常是一个对TYPE的参考。ARG\_TYPE是一个传递给CArray的参数。

说明

CArray类支持与C语言的Array相似的数组，但是必要时可以动态压缩并扩展。数组索引

**从0开始**。可以决定是固定数组上界还是允许当添加元素时扩展当前的边界。内存对上界是连续地分配空间，甚至一些元素可为空。和CArray一样，CArray索引元素的访问时间是不变的，与数组大小无关。

**提示**  **(初始化很重要)**

**在使用一个数组之前，使用SetSize建立它的大小和为它分配内存**。如果不使用SetSize，则为数组添加元素就会引起频繁地重新分配和拷贝。频繁地重新分配和拷贝不但没有效率，而且导致内存碎片。

CArray预定义大小后，使用Add函数会在末尾添加；而vector预定义大小后，使用push\_back函数仍然会在初始添加！！！

//但要注意的是，此时动态数组类似一个“固定”数组，添加元素时更多使用time[0]=a;time[1]=b,....

//而不是直接使用time.Add()函数，因为会在末尾添加！此时指针指向数组尾部！！！

如果需要一堆数组中的个别数据，必须设置CDumpContext对象的深度为1或更大。

此类的某成员函数调用全局帮助函数，它必须为CArray的大多数使用而定制。请参阅宏和全局量部分中的“类收集帮助器”。当从一个CArray对象中移去元素时，帮助函数DestructElements被调用。当添加元素时，帮助函数ConstructElements被调用。数组类的派生与列表的派生相似。如果要了解关于使用CArray的更多信息，请参阅联机文档“Visual C++程序员指南”中的“收集”。

#include”afxtempl.h”

请参阅 CObArray,DestructElement,ConstructElements,Collection ClassHelpers

CArray类的成员

构造函数

Carray构造一个空数组

属性

GetSize获得此数组中的元素数

GetUpperBound返回最大的有效索引值

SetSize设置包含在此数组中的元素数

操作

FreeExtra释放大于当前上界的未使用的内存

RemoveAll从此数组移去所有元素

元素访问

GetAt返回在给定索引上的值，不需特别使用 ，**使用[ ]即可；**

SetAt设定一个给定索引的值；数组不允许扩展 ，不需特别使用 ，**使用[ ]即可；**

ElementAt返回一个对数组中元素指针的临时参考

GetData允许对数组中的元素访问。可以为NULL

扩展数组

SetAtGrow为一个给定索引设置值；如果必要，扩展数组

Add在数组的末尾添加元素；如果必要，扩展数组

Append在数组上附加另一个数组；如果必要，扩展数组

Copy把另一个数组拷贝到数组上；如果必要，扩展数组

插入/移去

InsertAt在指定的索引上插入一个元素（或另一个数组中的所有元素）

RemoveAt在指定的索引上移去一个元素

运算符

**[ ]在特定索引上设置或获取元素 ，用法与一般数组一样，非常方便！**

## 3.具体的用法以及函数调用:

          3.1 如何得到向量中的元素?其用法和数组一样:

          例如:

          vector <int \*> a

          int b = 5;

          a.push\_back(b);//该函数下面有详解

          cout<<a[0];       //输出结果为5

1.push\_back   在数组的最后添加一个数据  
2.pop\_back    去掉数组的最后一个数据   
3.at                得到编号位置的数据  
4.begin           得到数组头的指针  
5.end             得到数组的最后一个单元+1的指针  
6．front        得到数组头的引用  
7.back            得到数组的最后一个单元的引用  
8.max\_size     得到vector最大可以是多大  
9.capacity       当前vector分配的大小  
10.size           当前使用数据的大小  
11.resize         改变当前使用数据的大小，如果它比当前使用的大，者填充默认值  
12.reserve      改变当前vecotr所分配空间的大小  
13.erase         删除指针指向的数据项  
14.clear          清空当前的vector  
15.rbegin        将vector反转后的开始指针返回(其实就是原来的end-1)  
16.rend          将vector反转后的结束指针返回(其实就是原来的begin-1)  
17.empty        判断vector是否为空  
18.swap         与另一个vector交换数据

        3.2  详细的函数实现功能：其中**vector<int> c.**

                             c.clear()         移除容器中所有数据。

                             c.empty()         判断容器是否为空。

                             c.erase(pos)        删除pos位置的数据

                             c.erase(beg,end) 删除[beg,end)区间的数据

                             c.front()         传回第一个数据。

                             c.insert(pos,elem)  在pos位置插入一个elem拷贝

                             c.pop\_back()     删除最后一个数据。

                             c.push\_back(elem) 在尾部加入一个数据。

                             c.resize(num)     重新设置该容器的大小

                             c.size()         回容器中实际数据的个数。

                             c.begin()           返回指向容器第一个元素的迭代器

                             c.end()             返回指向容器最后一个元素的迭代器

# 10.Vector与List用法（动态数组）

STL就是Standard Template Library，标准模板库。这可能是一个历史上最令人兴奋的工具的最无聊的术语。从根本上说，STL是一些“容器”的集合，这些“容器”有list,vector,set,map等。

CArray预定义大小后，使用Add函数会在末尾添加；而vector预定义大小后，使用push\_back函数仍然会在初始添加！！！

 vector(向量): C++中的一种数据结构,确切的说是一个类.它相当于一个动态的数组,当程序员无法知道自己需要的数组的规模多大时,用其来解决问题可以达到最大节约空间的目的.

   用法:

   1.文件包含:

           首先在程序开头处加上#include<vector>以包含所需要的类文件vector

          还有一定要加上using namespace std;

          2.变量声明:

               2.1 例:声明一个int向量以替代一维的数组:vector <int> a;(等于声明了一个int数组a[],大小没有指定,可以动态的向里面添加删除)。

               2.2 例:用vector代替二维数组.其实只要声明一个一维数组向量即可,而一个数组的名字其实代表的是它的首地址,所以只要声明一个地址的向量即可,即:vector <int \*> a.同理想用向量代替三维数组也是一样,vector <int\*\*>a;再往上面依此类推.

## 4.内存管理与效率

      1》使用reserve()函数提前设定容量大小，避免多次容量扩充操作导致效率低下。

        关于STL容器，最令人称赞的特性之一就是是只要不超过它们的最大大小，它们就可以自动增长到足以容纳你放进去的数据。（要知道这个最大值，只要调用名叫max\_size的成员函数。）对于vector和string，如果需要更多空间，就以类似realloc的思想来增长大小。vector容器支持随机访问，因此为了提高效率，它内部使用动态数组的方式实现的。在通过 reserve() 来申请特定大小的时候总是按指数边界来增大其内部缓冲区。当进行insert或push\_back等增加元素的操作时，如果此时动态数组的内存不够用，就要动态的重新分配当前大小的1.5~2倍的新内存区，再把原数组的内容复制过去。所以，在一般情况下，其访问速度同一般数组，只有在重新分配发生时，其性能才会下降。正如上面的代码告诉你的那样。而进行pop\_back操作时，capacity并不会因为vector容器里的元素减少而有所下降，还会维持操作之前的大小。对于vector容器来说，如果有大量的数据需要进行push\_back，应当使用reserve()函数提前设定其容量大小，否则会出现许多次容量扩充操作，导致效率低下。

      reserve成员函数允许你最小化必须进行的重新分配的次数，因而可以避免真分配的开销和迭代器/指针/引用失效。但在我解释reserve为什么可以那么做之前，让我简要介绍有时候令人困惑的四个相关成员函数。在标准容器中，只有vector和string提供了所有这些函数。

(1) size()告诉你容器中有多少元素。它没有告诉你容器为它容纳的元素分配了多少内存。   
(2) capacity()告诉你容器在它已经分配的内存中可以容纳多少元素。那是容器在那块内存中总共可以容纳多少元素，而不是还可以容纳多少元素。如果你想知道一个vector或string中有多少没有被占用的内存，你必须从capacity()中减去size()。如果size和capacity返回同样的值，容器中就没有剩余空间了，而下一次插入（通过insert或push\_back等）会引发上面的重新分配步骤。  
(3) resize([**Container**](http://lib.csdn.net/base/docker)::size\_type n)强制把容器改为容纳n个元素。调用resize之后，size将会返回n。如果n小于当前大小，容器尾部的元素会被销毁。如果n大于当前大小，新默认构造的元素会添加到容器尾部。如果n大于当前容量，在元素加入之前会发生重新分配。  
(4) reserve(Container::size\_type n)强制容器把它的容量改为至少n，提供的n不小于当前大小。这一般强迫进行一次重新分配，因为容量需要增加。（如果n小于当前容量，vector忽略它，这个调用什么都不做，string可能把它的容量减少为size()和n中大的数，但string的大小没有改变。在我的经验中，使用reserve来从一个string中修整多余容量一般不如使用“交换技巧”，那是条款17的主题。）

     这个简介表示了只要有元素需要插入而且容器的容量不足时就会发生重新分配（包括它们维护的原始内存分配和回收，对象的拷贝和析构和迭代器、指针和引用的失效）。所以，避免重新分配的关键是使用reserve尽快把容器的容量设置为足够大，最好在容器被构造之后立刻进行。

例如，假定你想建立一个容纳1-1000值的vector<int>。没有使用reserve，你可以像这样来做：

vector<int> v;  
for (int i = 1; i <= 1000; ++i) v.push\_back(i);  
在大多数STL实现中，这段代码在循环过程中将会导致2到10次重新分配。（10这个数没什么奇怪的。记住vector在重新分配发生时一般把容量翻倍，而1000约等于210。）

把代码改为使用reserve，我们得到这个：

vector<int> v;  
v.reserve(1000);  
for (int i = 1; i <= 1000; ++i) v.push\_back(i);  
这在循环中不会发生重新分配。

在大小和容量之间的关系让我们可以预言什么时候插入将引起vector或string执行重新分配，而且，可以预言什么时候插入会使指向容器中的迭代器、指针和引用失效。例如，给出这段代码，

string s;  
...  
if (s.size() < s.capacity()) {  
s.push\_back('x');  
}  
push\_back的调用不会使指向这个string中的迭代器、指针或引用失效，因为string的容量保证大于它的大小。如果不是执行push\_back，代码在string的任意位置进行一个insert，我们仍然可以保证在插入期间没有发生重新分配，但是，与伴随string插入时迭代器失效的一般规则一致，所有从插入位置到string结尾的迭代器/指针/引用将失效。

回到本条款的主旨，通常有两情况使用reserve来避免不必要的重新分配。第一个可用的情况是当你确切或者大约知道有多少元素将最后出现在容器中。那样的话，就像上面的vector代码，你只是提前reserve适当数量的空间。第二种情况是保留你可能需要的最大的空间，然后，一旦你添加完全部数据，修整掉任何多余的容量。

       2》使用“交换技巧”来修整vector过剩空间/内存

      有一种方法来把它从曾经最大的容量减少到它现在需要的容量。这样减少容量的方法常常被称为“收缩到合适（shrink to fit）”。该方法只需一条语句：vector<int>(ivec).swap(ivec);  
表达式vector<int>(ivec)建立一个临时vector，它是ivec的一份拷贝：vector的拷贝构造函数做了这个工作。但是，vector的拷贝构造函数只分配拷贝的元素需要的内存，所以这个临时vector没有多余的容量。然后我们让临时vector和ivec交换数据，这时我们完成了，ivec只有临时变量的修整过的容量，而这个临时变量则持有了曾经在ivec中的没用到的过剩容量。在这里（这个语句结尾），临时vector被销毁，因此释放了以前ivec使用的内存，收缩到合适。

     3》用swap方法强行释放STL Vector所占内存

template < class T> void ClearVector( vector<T>& v )  
{   
    vector<T>vtTemp;  
    vtTemp.swap( v );  
}   
如   
    vector<int> v ;  
    nums.push\_back(1);  
    nums.push\_back(3);  
    nums.push\_back(2);  
    nums.push\_back(4);  
    vector<int>().swap(v);

/\* 或者v.swap(vector<int>()); \*/

/\*或者{ vector<int> tmp = v;   v.swap(tmp);   }; //加大括号{ }是让tmp退出{ }时自动析构\*/

## **5.Vector 内存管理成员函数的**

行为[**测试**](http://lib.csdn.net/base/softwaretest)

       C++ STL的vector使用非常广泛，但是对其内存的管理模型一直有多种猜测，下面用实例代码测试来了解其内存管理方式，测试代码如下：

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

int main()  
{  
vector<int> iVec;  
cout << "容器 大小为: " << iVec.size() << endl;  
cout << "容器 容量为: " << iVec.capacity() << endl; //1个元素， 容器容量为1

iVec.push\_back(1);  
cout << "容器 大小为: " << iVec.size() << endl;  
cout << "容器 容量为: " << iVec.capacity() << endl; //2个元素， 容器容量为2

iVec.push\_back(2);  
cout << "容器 大小为: " << iVec.size() << endl;  
cout << "容器 容量为: " << iVec.capacity() << endl; //3个元素， 容器容量为4

iVec.push\_back(3);  
cout << "容器 大小为: " << iVec.size() << endl;  
cout << "容器 容量为: " << iVec.capacity() << endl; //4个元素， 容器容量为4

iVec.push\_back(4);  
iVec.push\_back(5);  
cout << "容器 大小为: " << iVec.size() << endl;  
cout << "容器 容量为: " << iVec.capacity() << endl; //5个元素， 容器容量为8

iVec.push\_back(6);  
cout << "容器 大小为: " << iVec.size() << endl;  
cout << "容器 容量为: " << iVec.capacity() << endl; //6个元素， 容器容量为8

iVec.push\_back(7);  
cout << "容器 大小为: " << iVec.size() << endl;  
cout << "容器 容量为: " << iVec.capacity() << endl; //7个元素， 容器容量为8

iVec.push\_back(8);  
cout << "容器 大小为: " << iVec.size() << endl;  
cout << "容器 容量为: " << iVec.capacity() << endl; //8个元素， 容器容量为8

iVec.push\_back(9);  
cout << "容器 大小为: " << iVec.size() << endl;  
cout << "容器 容量为: " << iVec.capacity() << endl; //9个元素， 容器容量为16  
/\* vs2005/8 容量增长不是翻倍的，如   
    9个元素   容量9   
    10个元素 容量13 \*/

/\* 测试effective stl中的特殊的交换 swap() \*/  
cout << "当前vector 的大小为: " << iVec.size() << endl;  
cout << "当前vector 的容量为: " << iVec.capacity() << endl;  
vector<int>(iVec).swap(iVec);

cout << "临时的vector<int>对象 的大小为: " << (vector<int>(iVec)).size() << endl;  
cout << "临时的vector<int>对象 的容量为: " << (vector<int>(iVec)).capacity() << endl;  
cout << "交换后，当前vector 的大小为: " << iVec.size() << endl;  
cout << "交换后，当前vector 的容量为: " << iVec.capacity() << endl;

return 0;  
}

## 6.vector的其他成员函数

        c.assign(beg,end)：将[beg; end)区间中的数据赋值给c。  
        c.assign(n,elem)：将n个elem的拷贝赋值给c。   
        c.at(idx)：传回索引idx所指的数据，如果idx越界，抛出out\_of\_range。   
        c.back()：传回最后一个数据，不检查这个数据是否存在。  
        c.front()：传回第一个数据。   
        get\_allocator：使用构造函数返回一个拷贝。   
        c.rbegin()：传回一个逆向队列的第一个数据。   
        c.rend()：传回一个逆向队列的最后一个数据的下一个位置。   
        c.~ vector <Elem>()：销毁所有数据，释放内存。

7.备注:在用vector的过程中的一些问题,特此列出讨论:

               1)

                    vector <int > a;

                    int  b = 5;

                    a.push\_back(b);

                    此时若对b另外赋值时不会影响a[0]的值

                2)

                    vector <int\*> a;  
                     int \*b;  
                     b= new int[4];  
                     b[0]=0;  
                     b[1]=1;  
                     b[2]=2;  
                     a.push\_back(b);  
                     delete b;          //释放b的地址空间  
                     for(int i=0 ; i <3 ; i++)  
                     {  
                           cout<<a[0][i]<<endl;  
                     }

                     此时输出的值并不是一开始b数组初始化的值,而是一些无法预计的值.

                    分析:根据1) 2)的结果,可以想到,在1)中,  往a向量中压入的是b的值,即a[0]=b,此时a[0]和b是存储在两个不同的地址中的.因此改变b的值不会影响a[0];而在2)中,因为是把一个地址(指针)压入向量a,即a[0]=b,因此释放了b的地址也就释放了a[0]的地址,因此a[0]数组中存放的数值也就不得而知了.

## 7.使用Vector实现二维数组

我们知道，C++的template参数是可以嵌套定义的，你可以这样定义一个模板的Instance

vector <vector <int> > array2(3);//注意> 和> 之间的空格。

这就是我们的关键，array2可以保存3个向量，向量的长度是可以改变的。array2[i]返回的是第i个向量。同理，array2[i][j]返回的是第i个向量中的第j个元素。

到这里，你可能会得意的说: "我明白了，很简单吗！ "。别急，还有一些细节问题：如下

vector <vector <int> > array2(3);

array2[1][2]=9;

我保证你的程序会segement failed,原因就是你没有指定向量的大小。用push\_back函数可以解决问题：array2[1].push\_back(9);但是好象不太爽。就不能用operator[]吗？答案是肯定的。不过要多加几个步骤，如下：

for(int i=0;i <3;i++)

array2[i].resize(3);

这样，你就定义了一个3X3的数组了（另一个3在 申明时定义的）。而且你可以随时改变它的大小。

若：

array.resize(5);

arry[3].resize(3);

arry[4].resize(3);

//现在是5X3的数组了

**List也是STL提供的一个动态容器，他主要用于处理动态大小的字符串数组，用法与vector类似**

list<string> Milkshakes;

list<string>::iterator MilkshakeIterator;//为Milkshakes实例容器定义一个iterator指针：MilkshakeIterator

Milkshakes.push\_back("Chocolate");

Milkshakes.push\_back("Strawberry");

Milkshakes.push\_front("Lime");

Milkshakes.push\_front("Vanilla");

// print the milkshakes打印

Milkshakes.push\_front("The Milkshake Menu");

Milkshakes.push\_back("\*\*\* Thats the end \*\*\*");

for (MilkshakeIterator=Milkshakes.begin(); MilkshakeIterator!=Milkshakes.end(); ++MilkshakeIterator)

{

// dereference the iterator to get the element

cout << \*MilkshakeIterator << endl; //输出

}

# 11.new和delete

主要是用于创建短暂的长度固定的数组，储存数据。

1.创建动态数组

数组类型的变量有三个重要的限制：数组长度固定不变，在编译时必须知道它的长度，数组只在定义它的块语句中存在。

int \*p=new int[100];//少使用

int \*p;

try{p=new int[100];}

catch(bad\_alloc e)

{//new内存分配不足时

cerr<<"Out of memory"<<endl;

exit(1);

}

C++中关于new和delete的使用

例子：

int \*p;

p=new int[5]; //新建一个暂时数组，指针P指向该数组

p[0] = 5; //将数据存放进数组之中

p[1] = 2;

delete [ ]p; //删除数组，释放内存

对于动态分配的数组，虽然长度是固定的，但是动态分配的数组不必在编译时知道其长度，可以（通常也是）在运行时才确定数组长度；同时，动态分配的数组一直存在直到程序显示释放为止，这样，我们就可以自己决定数组的存在与否了。

每个程序在执行时都占用一块可用的内存空间，用于存放动态分配的对象，此内存空间称为程序的自由存储区（free store）或堆（heap）.C++语言使用new和delete在自由存储区中分配存储空间。

动态分配数组时，只需指定类型和数组长度，不必为数组对象命名，new表达式返回指向新分配数组的第一个元素的指针：

int \*pia=new int[10];

此new表达式分配了一个含有10个int型元素的数组，并返回指向该数组第一个元素的指针。在自由存储区创建的数组对象是没有名字的，程序员只能通过其地址间接地访问堆中的对象。

动态分配数组时，如果数组元素具有类类型，将使用该类的默认构造函数实现初始化；如果数组元素是内置类型，则无初始化：

string \*psa = new string[10]; // 调用string类的默认构造函数依次初始化数组中的每个元素。

int \*pis = new int[10]; //无初始化值

可使用跟在数组长度后面的一对空圆括号，对数组元素作值初始化：

int \*pis = new int[10](); //数组元素都设置为0

之所以要动态分配数组，往往是由于编译时并不知道数组的长度。C++中允许动态分配空数组：

size\_t n = get\_size();

int \*p = new int[n];

for(int \*q=p;q!=p+n;++q)

『/\*处理数组元素的相关代码\*/』

在上面的例子中，只有在程序运行时才能确定n的值。如果n的值为0，代码依然正确执行。C++中虽然不允许定义长度为0的数组变量，但是明确指出，调用new动态创建长度为0的数组是合法的。

char \*cp = new char[10];

用new动态创建长度为0的数组时，new返回有效的非零指针，但不能进行解引用，因为它毕竟没有指向任何元素。

2. 动态创建单个对象

动态创建对象时，只需指定其数据类型，不必为该对象命名，new表达式返回指向新创建对象的指针，我们通过该指针来访问此对象：

int \*pi = new int(1024);

string \*ps = new string(10, '9');

在C++中使用直接初始化语法规则初始化动态创建的对象，上面的表达式就使用该语法规则初始化了动态创建的对象。对于类类型的对象，则使用该类类型的构造函数初始化对象。可对动态创建的对象做值初始化，如下所示：

string \*ps = new string();

int \*pi = new int();

对于提供了默认构造函数的类类型，没有必要进行值初始化。如果没有显示初始化动态创建的对象，则对于类类型的对象，用该类的默认构造函数初始化，对于内置类型的对象则无初始化。

3. 动态空间的释放

动态分配的内存最后必须进行释放。如果不需要动态创建的数组，我们必须显示地将其占用的存储空间返还给程序的自由存储区。C++中使用delete [ ]表达式释放指针所指向的数组空间：

delete [ ] pis;

该语句释放上面所创建的动态int型数组所占有的存储空间。在关键字delete和指针之间的[ ]告诉编译器该指针指向的是自由存储区中的数组，而并非单个对象。对于表达式

delete pi;

该命令释放了pi指向的int型对象所占用的内存空间。删除指针后，该指针变成悬垂指针（dangling pointer）.悬垂指针指向曾今存放对象的内存但该对象已经不存在了。悬垂指针往往导致程序错误，而且很难检测出来。一旦删除了指针所指向的对象，立即将指针置为0，这样就非常清楚地表明指针不再指向任何对象。

# 12.数组的复制

与Matlab程序不同，C++中不能使用赋值语句A=B进行数组B的复制，方法有两种：

用数组名作参数传递，会改变数组元素的值，因此，若程序员需要保留原始数组的拷贝函数可以通过把参数类型声明为 const 来表明不希望改变数组元素。  
void putValues( const int[ 10 ] );

如：

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

int b[2];

b=a;

错误！

正确方法如下：

1. 元素循环复制

for(i=0;i<=1;i++)

{

b[i]=a[i];

}

1. #include”memory.h”

int b[2];

memcpy(b,a,sizeof(a));

# 13.typedef 定义结构体的特点

typedef struct tagTime //利用typedef进行结构体创建与通常结构体创建区别不大

{ // 但typedef主要应用于 \*\*\*\*结构体的嵌套\*\*\*\*

int year; //

int mon; //

int day;

string name;

}Time;

typedef struct tagNowTime

{

Time now;

string loc;

}NowTime;

# 14.CString类详解

与string类的区别：

（1）CString 类是微软的visual c++提供的MFC里面的一个类，所以只有支持MFC的工程才可以使用。如在linux上的工程就不能用CString了，只能用标准C++中的string类了。另外，因为string类是在c++标准库中，所以它被封装在了std命名空间中，使用之前需要声明using namespace std;而CString类并不在std命名空间中，因为它不是c++的标准库，只是微软的一个封装库。这点看来用string类的程序的移植性更好。

（2）string类既是一个标准c++的类库，同时也是STL（Standard Template Library,标准模版库）中的类库，所以支持Iterator操作。

（3）CString类和string类提供的方法接口并不完全相同，所以不要糊里糊涂的认为某个类中怎么没有另外一个类中的方法啊。:-)。。

（4）他们和char\*之间的转换方法也不一样。

CString 是一种很有用的数据类型。它们很大程度上简化了MFC中的许多操作，使得MFC在做字符串操作的时候方便了很多。不管怎样，使用CString有很多特殊的技巧，特别是对于纯C背景下走出来的程序员来说有点难以学习。这篇文章就来讨论这些技巧。

使用CString可以让你对字符串的操作更加直截了当。这篇文章不是CString的完全手册，但囊括了大部分常见基本问题。

这篇文章包括以下内容：

1.CString 对象的连接

2. 格式化字符串（包括 int 型转化为 CString ）

3. CString 型转化成 int 型

4. CString 型和 char\* 类型的相互转化

5. char\* 转化成 CString

6. CString 转化成 char\* 之一：使用LPCTSTR强制转化

7. CString 转化成 char\* 之二：使用CString对象的GetBuffer方法

8. CString 转化成 char\* 之三: 和控件的接口

9. CString 型转化成 BSTR 型；

10. BSTR 型转化成 CString 型；

11. VARIANT 型转化成 CString 型；

12. 载入字符串表资源；

13. CString 和临时对象；

14. CString 的效率；

15. 总结

下面分别讨论。

## 1、CString 对象的连接

　　能体现出 CString 类型方便性特点的一个方面就字符串的连接，使用 CString 类型，你能很方便地连接两个字符串，正如下面的例子：

CString gray("Gray");CString cat("Cat");CString graycat = gray + cat;

要比用下面的方法好得多：

char gray[] = "Gray";char cat[] = "Cat";char \* graycat = malloc(strlen(gray) + strlen(cat) + 1);strcpy(graycat, gray);strcat(graycat, cat);

## 2、格式化字符串

　　与其用 sprintf() 函数或 wsprintf() 函数来格式化一个字符串，还不如用 CString 对象的Format()方法：

CString s;s.Format(\_T("The total is %d"), total);

　　用这种方法的好处是你不用担心用来存放格式化后数据的缓冲区是否足够大，这些工作由CString类替你完成。

　　格式化是一种把其它不是字符串类型的数据转化为CString类型的最常用技巧，比如，把一个整数转化成CString类型，可用如下方法：

CString s;s.Format(\_T("%d"), total);

　　我总是对我的字符串使用\_T()宏，这是为了让我的代码至少有Unicode的意识，当然，关于Unicode的话题不在这篇文章的讨论范围。\_T()宏在8位字符环境下是如下定义的：

#define \_T(x) x // 非Unicode版本（non-Unicode version）

而在Unicode环境下是如下定义的：

#define \_T(x) L##x // Unicode版本（Unicode version）

所以在Unicode环境下，它的效果就相当于：

s.Format(L"%d", total);

　　如果你认为你的程序可能在Unicode的环境下运行，那么开始在意用 Unicode 编码。比如说，不要用 sizeof() 操作符来获得字符串的长度，因为在Unicode环境下就会有2倍的误差。我们可以用一些方法来隐藏Unicode的一些细节，比如在我需要获得字符长度的时候，我会用一个叫做DIM的宏，这个宏是在我的dim.h文件中定义的，我会在我写的所有程序中都包含这个文件：

#define DIM(x) ( sizeof((x)) / sizeof((x)[0]) )

　　这个宏不仅可以用来解决Unicode的字符串长度的问题，也可以用在编译时定义的表格上，它可以获得表格的项数，如下：

class Whatever { ... };Whatever data[] = { { ... }, ... { ... },};for(int i = 0; i < DIM(data); i++) // 扫描表格寻找匹配项。

　　这里要提醒你的就是一定要注意那些在参数中需要真实字节数的API函数调用，如果你传递字符个数给它，它将不能正常工作。如下：

TCHAR data[20];lstrcpyn(data, longstring, sizeof(data) - 1); // WRONG!lstrcpyn(data, longstring, DIM(data) - 1); // RIGHTWriteFile(f, data, DIM(data), &bytesWritten, NULL); // WRONG!WriteFile(f, data, sizeof(data), &bytesWritten, NULL); // RIGHT

造成以上原因是因为lstrcpyn需要一个字符个数作为参数，但是WriteFile却需要字节数作为参数。

同样需要注意的是有时候需要写出数据的所有内容。如果你仅仅只想写出数据的真实长度，你可能会认为你应该这样做：

WriteFile(f, data, lstrlen(data), &bytesWritten, NULL); // WRONG

但是在Unicode环境下，它不会正常工作。正确的做法应该是这样：

WriteFile(f, data, lstrlen(data) \* sizeof(TCHAR), &bytesWritten, NULL); // RIGHT

　　因为WriteFile需要的是一个以字节为单位的长度。（可能有些人会想“在非Unicode的环境下运行这行代码，就意味着总是在做一个多余的乘1 操作，这样不会降低程序的效率吗？”这种想法是多余的，你必须要了解编译器实际上做了什么，没有哪一个C或C++编译器会把这种无聊的乘1操作留在代码中。在Unicode环境下运行的时候，你也不必担心那个乘2操作会降低程序的效率，记住，这只是一个左移一位的操作而已，编译器也很乐意为你做这种替换。）

　　使用\_T宏并不是意味着你已经创建了一个Unicode的程序，你只是创建了一个有Unicode意识的程序而已。如果你在默认的 8-bit模式下编译你的程序的话，得到的将是一个普通的8-bit的应用程序（这里的8-bit指的只是8位的字符编码，并不是指8位的计算机系统）；当你在Unicode环境下编译你的程序时，你才会得到一个Unicode的程序。记住，CString 在 Unicode 环境下，里面包含的可都是16位的字符哦。

## 3、CString 型转化成 int 型

　　把 CString 类型的数据转化成整数类型最简单的方法就是使用标准的字符串到整数转换例程。

　　虽然通常你怀疑使用\_atoi()函数是一个好的选择，它也很少会是一个正确的选择。如果你准备使用 Unicode 字符，你应该用\_ttoi()，它在 ANSI 编码系统中被编译成\_atoi()，而在 Unicode 编码系统中编译成\_wtoi()。你也可以考虑使用\_tcstoul()或者\_tcstol()，它们都能把字符串转化成任意进制的长整数（如二进制、八进制、十进制或十六进制），不同点在于前者转化后的数据是无符号的（unsigned），而后者相反。看下面的例子：

CString hex = \_T("FAB");CString decimal = \_T("4011");ASSERT(\_tcstoul(hex, 0, 16) == \_ttoi(decimal));

## 4、CString 型和 char\* 类型的相互转化

　　这是初学者使用 CString 时最常见的问题。有了 C++ 的帮助，很多问题你不需要深入的去考虑它，直接拿来用就行了，但是如果你不能深入了解它的运行机制，又会有很多问题让你迷惑，特别是有些看起来没有问题的代码，却偏偏不能正常工作。

比如，你会奇怪为什么不能写向下面这样的代码呢：

CString graycat = "Gray" + "Cat";

或者这样：

CString graycat("Gray" + "Cat");

　　事实上，编译器将抱怨上面的这些尝试。为什么呢？因为针对CString 和 LPCTSTR数据类型的各种各样的组合，“ +” 运算符被定义成一个重载操作符。而不是两个 LPCTSTR 数据类型，它是底层数据类型。你不能对基本数据（如 int、char 或者 char\*）类型重载 C++ 的运算符。你可以象下面这样做：

CString graycat = CString("Gray") + CString("Cat");

或者这样：

CString graycat = CString("Gray") + "Cat";

研究一番就会发现：“ +”总是使用在至少有一个 CString 对象和一个 LPCSTR 的场合。

注意，编写有 Unicode 意识的代码总是一件好事，比如：

CString graycat = CString(\_T("Gray")) + \_T("Cat");

这将使得你的代码可以直接移植。

char\* 转化为 CString

　　现在你有一个 char\* 类型的数据，或者说一个字符串。怎么样创建 CString 对象呢？这里有一些例子：

char \* p = "This is a test";

或者象下面这样更具有 Unicode 意识：

TCHAR \* p = \_T("This is a test")

或

LPTSTR p = \_T("This is a test");

你可以使用下面任意一种写法：

CString s = "This is a test"; // 8-bit onlyCString s = \_T("This is a test"); // Unicode-awareCString s("This is a test"); // 8-bit onlyCString s(\_T("This is a test")); // Unicode-awareCString s = p;CString s(p);

　用这些方法可以轻松将常量字符串或指针转换成 CString。需要注意的是，字符的赋值总是被拷贝到 CString 对象中去的，所以你可以象下面这样操作：

TCHAR \* p = \_T("Gray");CString s(p);p = \_T("Cat");s += p;

结果字符串肯定是“GrayCat”。

CString 类还有几个其它的构造函数，但是这里我们不考虑它，如果你有兴趣可以自己查看相关文档。

事实上，CString 类的构造函数比我展示的要复杂，比如：

CString s = "This is a test";

　　这是很草率的编码，但是实际上它在 Unicode 环境下能编译通过。它在运行时调用构造函数的 MultiByteToWideChar 操作将 8 位字符串转换成 16 位字符串。不管怎样，如果 char \* 指针是网络上传输的 8 位数据，这种转换是很有用的。

CString 转化成 char\* 之一：强制类型转换为 LPCTSTR；

　　这是一种略微硬性的转换，有关“正确”的做法，人们在认识上还存在许多混乱，正确的使用方法有很多，但错误的使用方法可能与正确的使用方法一样多。

　　我们首先要了解 CString 是一种很特殊的 C++ 对象，它里面包含了三个值：一个指向某个数据缓冲区的指针、一个是该缓冲中有效的字符记数以及一个缓冲区长度。有效字符数的大小可以是从0到该缓冲最大长度值减1之间的任何数（因为字符串结尾有一个NULL字符）。字符记数和缓冲区长度被巧妙隐藏。

　　除非你做一些特殊的操作，否则你不可能知道给CString对象分配的缓冲区的长度。这样，即使你获得了该0缓冲的地址，你也无法更改其中的内容，不能截短字符串，也绝对没有办法加长它的内容，否则第一时间就会看到溢出。

　　LPCTSTR 操作符（或者更明确地说就是 TCHAR \* 操作符）在 CString 类中被重载了，该操作符的定义是返回缓冲区的地址，因此，如果你需要一个指向 CString 的 字符串指针的话，可以这样做：

CString s("GrayCat");LPCTSTR p = s;

　　它可以正确地运行。这是由C语言的强制类型转化规则实现的。当需要强制类型转化时，C++规测容许这种选择。比如，你可以将（浮点数）定义为将某个复数（有一对浮点数）进行强制类型转换后只返回该复数的第一个浮点数（也就是其实部）。可以象下面这样：

Complex c(1.2f, 4.8f);float realpart = c;

如果(float)操作符定义正确的话，那么实部的的值应该是1.2。

　　这种强制转化适合所有这种情况，例如，任何带有 LPCTSTR 类型参数的函数都会强制执行这种转换。于是，你可能有这样一个函数（也许在某个你买来的DLL中）：

BOOL DoSomethingCool(LPCTSTR s);

你象下面这样调用它：

CString file("c:\\myfiles\\coolstuff")BOOL result = DoSomethingCool(file);

　　它能正确运行。因为 DoSomethingCool 函数已经说明了需要一个 LPCTSTR 类型的参数，因此 LPCTSTR 被应用于该参数，在 MFC 中就是返回的串地址。

如果你要格式化字符串怎么办呢？

CString graycat("GrayCat");CString s;s.Format("Mew! I love %s", graycat);

　　注意由于在可变参数列表中的值（在函数说明中是以“...”表示的）并没有隐含一个强制类型转换操作符。你会得到什么结果呢？

　　一个令人惊讶的结果，我们得到的实际结果串是：

"Mew! I love GrayCat"。

　　因为 MFC 的设计者们在设计 CString 数据类型时非常小心， CString 类型表达式求值后指向了字符串，所以这里看不到任何象 Format 或 sprintf 中的强制类型转换，你仍然可以得到正确的行为。描述 CString 的附加数据实际上在 CString 名义地址之后。

　　有一件事情你是不能做的，那就是修改字符串。比如，你可能会尝试用“,”代替“.”（不要做这样的，如果你在乎国际化问题，你应该使用十进制转换的 National Language Support 特性，），下面是个简单的例子：

CString v("1.00"); // 货币金额，两位小数LPCTSTR p = v;p[lstrlen(p) - 3] = '','';

　　这时编译器会报错，因为你赋值了一个常量串。如果你做如下尝试，编译器也会错：

strcat(p, "each");

　　因为 strcat 的第一个参数应该是 LPTSTR 类型的数据，而你却给了一个 LPCTSTR。

　　不要试图钻这个错误消息的牛角尖，这只会使你自己陷入麻烦！

　　原因是缓冲有一个计数，它是不可存取的（它位于 CString 地址之下的一个隐藏区域），如果你改变这个串，缓冲中的字符计数不会反映所做的修改。此外，如果字符串长度恰好是该字符串物理限制的长度（梢后还会讲到这个问题），那么扩展该字符串将改写缓冲以外的任何数据，那是你无权进行写操作的内存（不对吗？），你会毁换坏不属于你的内存。这是应用程序真正的死亡处方。

CString转化成char\* 之二：使用 CString 对象的 GetBuffer 方法；

　　如果你需要修改 CString 中的内容，它有一个特殊的方法可以使用，那就是 GetBuffer，它的作用是返回一个可写的缓冲指针。如果你只是打算修改字符或者截短字符串，你完全可以这样做：

CString s(\_T("File.ext"));LPTSTR p = s.GetBuffer();LPTSTR dot = strchr(p, ''.''); // OK, should have used s.Find...if(p != NULL)\*p = \_T(''\0'');s.ReleaseBuffer();

　　这是 GetBuffer 的第一种用法，也是最简单的一种，不用给它传递参数，它使用默认值 0，意思是：“给我这个字符串的指针，我保证不加长它”。当你调用 ReleaseBuffer 时，字符串的实际长度会被重新计算，然后存入 CString 对象中。

　　必须强调一点，在 GetBuffer 和 ReleaseBuffer 之间这个范围，一定不能使用你要操作的这个缓冲的 CString 对象的任何方法。因为 ReleaseBuffer 被调用之前，该 CString 对象的完整性得不到保障。研究以下代码：

CString s(...);LPTSTR p = s.GetBuffer();//... 这个指针 p 发生了很多事情int n = s.GetLength(); // 很糟D!!!!! 有可能给出错误的答案!!!s.TrimRight(); // 很糟!!!!! 不能保证能正常工作!!!!s.ReleaseBuffer(); // 现在应该 OKint m = s.GetLength(); // 这个结果可以保证是正确的。s.TrimRight(); // 将正常工作。

　　假设你想增加字符串的长度，你首先要知道这个字符串可能会有多长，好比是声明字符串数组的时候用：

char buffer[1024];

表示 1024 个字符空间足以让你做任何想做得事情。在 CString 中与之意义相等的表示法：

LPTSTR p = s.GetBuffer(1024);

　　调用这个函数后，你不仅获得了字符串缓冲区的指针，而且同时还获得了长度至少为 1024 个字符的空间（注意，我说的是“字符”，而不是“字节”，因为 CString 是以隐含方式感知 Unicode 的）。

　　同时，还应该注意的是，如果你有一个常量串指针，这个串本身的值被存储在只读内存中，如果试图存储它，即使你已经调用了 GetBuffer ，并获得一个只读内存的指针，存入操作会失败，并报告存取错误。我没有在 CString 上证明这一点，但我看到过大把的 C 程序员经常犯这个错误。

　　C 程序员有一个通病是分配一个固定长度的缓冲，对它进行 sprintf 操作，然后将它赋值给一个 CString：

char buffer[256];sprintf(buffer, "%......", args, ...); // ... 部分省略许多细节CString s = buffer;

虽然更好的形式可以这么做：

CString s;s.Format(\_T("%...."), args, ...);

如果你的字符串长度万一超过 256 个字符的时候，不会破坏堆栈。

　　另外一个常见的错误是：既然固定大小的内存不工作，那么就采用动态分配字节，这种做法弊端更大：

int len = lstrlen(parm1) + 13 lstrlen(parm2) + 10 + 100;char \* buffer = new char[len];sprintf(buffer, "%s is equal to %s, valid data", parm1, parm2);CString s = buffer;......delete [] buffer;

它可以能被简单地写成：

CString s;s.Format(\_T("%s is equal to %s, valid data"), parm1, parm2);

　　需要注意 sprintf 例子都不是 Unicode 就绪的，尽管你可以使用 tsprintf 以及用 \_T() 来包围格式化字符串，但是基本思路仍然是在走弯路，这这样很容易出错。

CString to char \* 之三：和控件的接口；

　　我们经常需要把一个 CString 的值传递给一个控件，比如，CTreeCtrl。MFC为我们提供了很多便利来重载这个操作，但是在大多数情况下，你使用“原始”形式的更新，因此需要将墨某个串指针存储到 TVINSERTITEMSTRUCT 结构的 TVITEM 成员中。如下：

TVINSERTITEMSTRUCT tvi;CString s;// ... 为s赋一些值。tvi.item.pszText = s; // Compiler yells at you here// ... 填写tvi的其他域HTREEITEM ti = c\_MyTree.InsertItem(&tvi);

　　为什么编译器会报错呢？明明看起来很完美的用法啊！但是事实上如果你看看 TVITEM 结构的定义你就会明白，在 TVITEM 结构中 pszText 成员的声明如下：

LPTSTR pszText;int cchTextMax;

　　因此，赋值不是赋给一个 LPCTSTR 类型的变量，而且编译器无法知道如何将赋值语句右边强制转换成 LPCTSTR。好吧，你说，那我就改成这样：

tvi.item.pszText = (LPCTSTR)s; //编译器依然会报错。

　　编译器之所以依然报错是因为你试图把一个 LPCTSTR 类型的变量赋值给一个 LPTSTR 类型的变量，这种操作在C或C++中是被禁止的。你不能用这种方法来滥用常量指针与非常量指针概念，否则，会扰乱编译器的优化机制，使之不知如何优化你的程序。比如，如果你这么做：

const int i = ...;//... do lots of stuff... = a[i]; // usage 1// ... lots more stuff... = a[i]; // usage 2

　　那么，编译器会以为既然 i 是 const ，所以 usage1和usage2的值是相同的，并且它甚至能事先计算好 usage1 处的 a[i] 的地址，然后保留着在后面的 usage2 处使用，而不是重新计算。如果你按如下方式写的话：

const int i = ...;int \* p = &i;//... do lots of stuff... = a[i]; // usage 1// ... lots more stuff(\*p)++; // mess over compiler''s assumption// ... and other stuff... = a[i]; // usage 2

　　编译器将认为 i 是常量，从而 a[i] 的位置也是常量，这样间接地破坏了先前的假设。因此，你的程序将会在 debug 编译模式（没有优化）和 release 编译模式（完全优化）中反映出不同的行为，这种情况可不好，所以当你试图把指向 i 的指针赋值给一个可修改的引用时，会被编译器诊断为这是一种伪造。这就是为什么（LPCTSTR）强制类型转化不起作用的原因。

　　为什么不把该成员声明成 LPCTSTR 类型呢？因为这个结构被用于读写控件。当你向控件写数据时，文本指针实际上被当成 LPCTSTR，而当你从控件读数据时，你必须有一个可写的字符串。这个结构无法区分它是用来读还是用来写。

因此，你会常常在我的代码中看到如下的用法：

tvi.item.pszText = (LPTSTR)(LPCTSTR)s;

　　它把 CString 强制类型转化成 LPCTSTR，也就是说先获得改字符串的地址，然后再强制类型转化成 LPTSTR，以便可以对之进行赋值操作。 注意这只有在使用 Set 或 Insert 之类的方法才有效！如果你试图获取数据，则不能这么做。

　　如果你打算获取存储在控件中的数据，则方法稍有不同，例如，对某个 CTreeCtrl 使用 GetItem 方法，我想获取项目的文本。我知道这些文本的长度不会超过 MY\_LIMIT，因此我可以这样写：

TVITEM tvi;// ... assorted initialization of other fields of tvitvi.pszText = s.GetBuffer(MY\_LIMIT);tvi.cchTextMax = MY\_LIMIT;c\_MyTree.GetItem(&tvi);s.ReleaseBuffer();

　　可以看出来，其实 上面的代码对所有类型的 Set 方法都适用，但是并不需要这么做，因为所有的类 Set 方法（包括 Insert方法）不会改变字符串的内容。但是当你需要写 CString 对象时，必须保证缓冲是可写的，这正是 GetBuffer 所做的事情。再次强调： 一旦做了一次 GetBuffer 调用，那么在调用 ReleaseBuffer 之前不要对这个 CString 对象做任何操作。

## 5、CString 型转化成 BSTR 型

　　当我们使用 ActiveX 控件编程时，经常需要用到将某个值表示成 BSTR 类型。BSTR 是一种记数字符串，Intel平台上的宽字符串（Unicode），并且可以包含嵌入的 NULL 字符。

你可以调用 CString 对象的 AllocSysString 方法将 CString 转化成 BSTR：

CString s;s = ... ; // whateverBSTR b = s.AllocSysString();

　　现在指针 b 指向的就是一个新分配的 BSTR 对象，该对象是 CString 的一个拷贝，包含终结 NULL字符。现在你可以将它传递给任何需要 BSTR 的接口。通常，BSTR 由接收它的组件来释放，如果你需要自己释放 BSTR 的话，可以这么做：

::SysFreeString(b);

　　对于如何表示传递给 ActiveX 控件的字符串，在微软内部曾一度争论不休，最后 Visual Basic 的人占了上风，BSTR（“Basic String”的首字母缩写）就是这场争论的结果。

## 6、BSTR 型转化成 CString 型

　 　由于 BSTR 是记数 Unicode 字符串，你可以用标准转换方法来创建 8 位的 CString。实际上，这是 CString 内建的功能。在 CString 中有特殊的构造函数可以把 ANSI 转化成 Unicode，也可以把Unicode 转化成 ANSI。你同样可以从 VARIANT 类型的变量中获得 BSTR 类型的字符串，VARIANT 类型是 由各种 COM 和 Automation (自动化)调用返回的类型。

例如，在一个ANSI程序中：

BSTR b;b = ...; // whateverCString s(b == NULL ? L"" : b)

　 　对于单个的 BSTR 串来说，这种用法可以工作得很好，这是因为 CString 有一个特殊的构造函数以LPCWSTR（BSTR正是这种类型）为参数，并将它转化成 ANSI 类型。专门检查是必须的，因为 BSTR 可能为空值，而 CString 的构造函数对于 NULL 值情况考虑的不是很周到，（感谢 Brian Ross 指出这一点!）。这种用法也只能处理包含 NUL 终结字符的单字符串；如果要转化含有多个 NULL 字符串，你得额外做一些工作才行。在 CString 中内嵌的 NULL 字符通常表现不尽如人意，应该尽量避免。

　　根据 C/C++ 规则，如果你有一个 LPWSTR，那么它别无选择，只能和 LPCWSTR 参数匹配。

在 Unicode 模式下，它的构造函数是：

CString::CString(LPCTSTR);

正如上面所表示的，在 ANSI 模式下，它有一个特殊的构造函数：

CString::CString(LPCWSTR); 它会调用一个内部的函数将 Unicode 字符串转换成 ANSI 字符串。（在Unicode模式下，有一个专门的构造函数，该函数有一个参数是LPCSTR类型——一个8位 ANSI 字符串指针，该函数将它加宽为 Unicode 的字符串！）再次强调：一定要检查 BSTR 的值是否为 NULL。

　　另外还有一个问题，正如上文提到的：BSTRs可以含有多个内嵌的NULL字符，但是 CString 的构造函数只能处理某个串中单个 NULL 字符。也就是说，如果串中含有嵌入的 NUL字节，CString 将会计算出错误的串长度。你必须自己处理它。如果你看看 strcore.cpp 中的构造函数，你会发现它们都调用了lstrlen，也就是计算字符串的长度。

　　注意从 Unicode 到 ANSI 的转换使用带专门参数的 ::WideCharToMultiByte，如果你不想使用这种默认的转换方式，则必须编写自己的转化代码。

　　如果你在 UNICODE 模式下编译代码，你可以简单地写成：

CString convert(BSTR b){ if(b == NULL) return CString(\_T("")); CString s(b); // in UNICODE mode return s;}

　　如果是 ANSI 模式，则需要更复杂的过程来转换。注意这个代码使用与 ::WideCharToMultiByte 相同的参数值。所以你只能在想要改变这些参数进行转换时使用该技术。例如，指定不同的默认字符，不同的标志集等。

CString convert(BSTR b){ CString s; if(b == NULL) return s; // empty for NULL BSTR#ifdef UNICODE s = b;#else LPSTR p = s.GetBuffer(SysStringLen(b) + 1); ::WideCharToMultiByte(CP\_ACP, // ANSI Code Page 0, // no flags b, // source widechar string -1, // assume NUL-terminated p, // target buffer SysStringLen(b)+1, // target buffer length NULL, // use system default char NULL); // don''t care if default used s.ReleaseBuffer();#endif return s;}

　　我并不担心如果 BSTR 包含没有映射到 8 位字符集的 Unicode 字符时会发生什么，因为我指定了::WideCharToMultiByte 的最后两个参数为 NULL。这就是你可能需要改变的地方。

## 7、VARIANT 型转化成 CString 型

　 　事实上，我从来没有这么做过，因为我没有用 COM/OLE/ActiveX 编写过程序。但是我在microsoft.public.vc.mfc 新闻组上看到了 Robert Quirk 的一篇帖子谈到了这种转化，我觉得把他的文章包含在我的文章里是不太好的做法，所以在这里多做一些解释和演示。如果和他的文章有相孛的地方可能是我的疏忽。

　　VARIANT 类型经常用来给 COM 对象传递参数，或者接收从 COM 对象返回的值。你也能自己编写返回 VARIANT 类型的方法，函数返回什么类型依赖可能（并且常常）方法的输入参数（比如，在自动化操作中，依赖与你调用哪个方法。IDispatch::Invoke 可能返回（通过其一个参数）一个包含有BYTE、WORD、float、double、date、BSTR 等等 VARIANT 类型的结果，（详见 MSDN 上的 VARIANT 结构的定义）。在下面的例子中，假设类型是一个BSTR的变体，也就是说在串中的值是通过 bsrtVal 来引用，其优点是在 ANSI 应用中，有一个构造函数会把 LPCWCHAR 引用的值转换为一个 CString（见 BSTR-to-CString 部分）。在 Unicode 模式中，将成为标准的 CString 构造函数，参见对缺省::WideCharToMultiByte 转换的告诫，以及你觉得是否可以接受（大多数情况下，你会满意的）。

VARIANT vaData;vaData = m\_com.YourMethodHere();ASSERT(vaData.vt == VT\_BSTR);CString strData(vaData.bstrVal);

你还可以根据 vt 域的不同来建立更通用的转换例程。为此你可能会考虑：

CString VariantToString(VARIANT \* va){ CString s; switch(va->vt) { case VT\_BSTR: return CString(vaData->bstrVal); case VT\_BSTR | VT\_BYREF: return CString(\*vaData->pbstrVal); case VT\_I4: s.Format(\_T("%d"), va->lVal); return s; case VT\_I4 | VT\_BYREF: s.Format(\_T("%d"), \*va->plVal); case VT\_R8: s.Format(\_T("%f"), va->dblVal); return s; ... 剩下的类型转换由读者自己完成 default: ASSERT(FALSE); // unknown VARIANT type (this ASSERT is optional) return CString(""); } }

## 8、载入字符串表资源

　　如果你想创建一个容易进行语言版本移植的应用程序，你就不能在你的源代码中直接包含本土语言字符串（下面这些例子我用的语言都是英语，因为我的本土语是英语），比如下面这种写法就很糟：

CString s = "There is an error";

　　你应该把你所有特定语言的字符串单独摆放（调试信息、在发布版本中不出现的信息除外）。这意味着向下面这样写比较好：

s.Format(\_T("%d - %s"), code, text);

　　在你的程序中，文字字符串不是语言敏感的。不管怎样，你必须很小心，不要使用下面这样的串：

// fmt is "Error in %s file %s"// readorwrite is "reading" or "writing"s.Format(fmt, readorwrite, filename);

　　这是我的切身体会。在我的第一个国际化的应用程序中我犯了这个错误，尽管我懂德语，知道在德语的语法中动词放在句子的最后面，我们的德国方面的发行人还是苦苦的抱怨他们不得不提取那些不可思议的德语错误提示信息然后重新格式化以让它们能正常工作。比较好的办法（也是我现在使用的办法）是使用两个字符串，一个用于读，一个用于写，在使用时加载合适的版本，使得它们对字符串参数是非敏感的。也就是说加载整个格式，而不是加载串“reading”， “writing”：

// fmt is "Error in reading file %s"// "Error in writing file %s"s.Format(fmt, filename);

　　一定要注意，如果你有好几个地方需要替换，你一定要保证替换后句子的结构不会出现问题，比如在英语中，可以是主语-宾语，主语-谓语，动词-宾语的结构等等。

　 　在这里，我们并不讨论 FormatMessage，其实它比 sprintf/Format 还要有优势，但是不太容易和CString 结合使用。解决这种问题的办法就是我们按照参数出现在参数表中的位置给参数取名字，这样在你输出的时候就不会把他们的位置排错了。

　　接下来我们讨论我们这些独立的字符串放在什么地方。我们可以把字符串的值放入资源文件中的一个称为 STRINGTABLE 的段中。过程如下：首先使用 Visual Studio 的资源编辑器创建一个字符串，然后给每一个字符串取一个ID，一般我们给它取名字都以 IDS\_开头。所以如果你有一个信息，你可以创建一个字符串资源然后取名为 IDS\_READING\_FILE，另外一个就取名为 IDS\_WRITING\_FILE。它们以下面的形式出现在你的 .rc 文件中：

STRINGTABLEIDS\_READING\_FILE "Reading file %s"IDS\_WRITING\_FILE "Writing file %s"END

注意：这些资源都以 Unicode 的格式保存，不管你是在什么环境下编译。他们在Win9x系统上也是以Unicode 的形式存在，虽然 Win9x 不能真正处理 Unicode。

然后你可以这样使用这些资源：

// 在使用资源串表之前，程序是这样写的：

CString fmt; if(...) fmt = "Reading file %s"; else fmt = "Writing file %s"; ... // much later CString s; s.Format(fmt, filename);

// 使用资源串表之后，程序这样写：

CString fmt; if(...) fmt.LoadString(IDS\_READING\_FILE); else fmt.LoadString(DS\_WRITING\_FILE); ... // much later CString s; s.Format(fmt, filename);

　　现在，你的代码可以移植到任何语 言中去。LoadString 方法需要一个字符串资源的 ID 作为参数，然后它从 STRINGTABLE 中取出它对应的字符串，赋值给 CString 对象。 CString 对象的构造函数还有一个更加聪明的特征可以简化 STRINGTABLE 的使用。这个用法在 CString::CString 的文档中没有指出，但是在构造函数的示例程序中使用了。（为什么这个特性没有成为正式文档的一部分，而是放在了一个例子中，我记不得了！）——【译者注：从这句话看，作者可能是CString的设计者。其实前面还有一句类似的话。说他没有对使用GetBuffer(0)获得的指针指向的地址是否可读做有效性检查】。这个特征就是：如果你将一个字符串资源的ID强制类型转换为 LPCTSTR，将会隐含调用 LoadString。因此，下面两个构造字符串的例子具有相同的效果，而且其 ASSERT 在debug模式下不会被触发：

CString s;s.LoadString(IDS\_WHATEVER);CString t( (LPCTSTR)IDS\_WHATEVER );ASSERT(s == t);//不会被触发，说明s和t是相同的。

　 　现在，你可能会想：这怎么可能工作呢？我们怎么能把 STRINGTABLE ID 转化成一个指针呢？很简单：所有的字符串 ID 都在1~65535这个范围内，也就是说，它所有的高位都是0，而我们在程序中所使用的指针是不可能小于65535的，因为程序的低 64K 内存永远也不可能存在的，如果你试图访问0x00000000到0x0000FFFF之间的内存，将会引发一个内存越界错误。所以说1~65535的值不可能是一个内存地址，所以我们可以用这些值来作为字符串资源的ID。

　　我倾向于使用 MAKEINTRESOURCE 宏显式地做这种转换。我认为这样可以让代码更加易于阅读。这是个只适合在 MFC 中使用的标准宏。你要记住，大多数的方法即可以接受一个 UINT 型的参数，也可以接受一个 LPCTSTR 型的参数，这是依赖 C++ 的重载功能做到的。C++重载函数带来的弊端就是造成所有的强制类型转化都需要显示声明。同样，你也可以给很多种结构只传递一个资源名。

CString s;s.LoadString(IDS\_WHATEVER);CString t( MAKEINTRESOURCE(IDS\_WHATEVER));ASSERT(s == t);

　　告诉你吧：我不仅只是在这里鼓吹，事实上我也是这么做的。在我的代码中，你几乎不可能找到一个字符串，当然，那些只是偶然在调试中出现的或者和语言无关的字符串除外。

## 9、CString 和临时对象

　　这是出现在 microsoft.public.vc.mfc 新闻组中的一个小问题，我简单的提一下，这个问题是有个程序员需要往注册表中写入一个字符串，他写道：

　 　我试着用 RegSetValueEx() 设置一个注册表键的值，但是它的结果总是令我困惑。当我用char[]声明一个变量时它能正常工作，但是当我用 CString 的时候，总是得到一些垃圾："葺葺...& Yacute;葺葺"为了确认是不是我的 CString 数据出了问题，我试着用 GetBuffer，然后强制转化成 char\*，LPCSTR。GetBuffer 返回的值是正确的，但是当我把它赋值给 char\* 时，它就变成垃圾了。以下是我的程序段：

char\* szName = GetName().GetBuffer(20);RegSetValueEx(hKey, "Name", 0, REG\_SZ, (CONST BYTE \*) szName, strlen (szName + 1));

这个 Name 字符串的长度小于 20，所以我不认为是 GetBuffer 的参数的问题。

真让人困惑，请帮帮我。

亲爱的 Frustrated，

你犯了一个相当微妙的错误，聪明反被聪明误，正确的代码应该象下面这样：

CString Name = GetName();RegSetValueEx(hKey, \_T("Name"), 0, REG\_SZ, (CONST BYTE \*) (LPCTSTR)Name, (Name.GetLength() + 1) \* sizeof(TCHAR));

　　为什么我写的代码能行而你写的就有问题呢？主要是因为当你调用 GetName 时返回的 CString 对象是一个临时对象。参见：《C++ Reference manual》§12.2

　　在一些环境中，编译器有必要创建一个临时对象，这样引入临时对象是依赖于实现的。如果编译器引入的这个临时对象所属的类有构造函数的话，编译器要确保这个类的构造函数被调用。同样的，如果这个类声明有析构函数的话，也要保证这个临时对象的析构函数被调用。

　　编译器必须保证这个临时对象被销毁了。被销毁的确切地点依赖于实现.....这个析构函数必须在退出创建该临时对象的范围之前被调用。

　　大部分的编译器是这样设计的：在临时对象被创建的代码的下一个执行步骤处隐含调用这个临时对象的析构函数，实现起来，一般都是在下一个分号处。因此，这个 CString 对象在 GetBuffer 调用之后就被析构了（顺便提一句，你没有理由给 GetBuffer 函数传递一个参数，而且没有使用ReleaseBuffer 也是不对的）。所以 GetBuffer 本来返回的是指向这个临时对象中字符串的地址的指针，但是当这个临时对象被析构后，这块内存就被释放了。然后 MFC 的调试内存分配器会重新为这块内存全部填上 0xDD，显示出来刚好就是“荨狈拧Ｔ谡飧鍪焙蚰阆蜃⒉岜碇行词荩址哪谌莸比蝗黄苹盗恕

　 　我们不应该立即把这个临时对象转化成 char\* 类型，应该先把它保存到一个 CString 对象中，这意味着把临时对象复制了一份，所以当临时的 CString 对象被析构了之后，这个 CString 对象中的值依然保存着。这个时候再向注册表中写数据就没有问题了。

　　此外，我的代码是具有 Unicode 意识的。那个操作注册表的函数需要一个字节大小，使用lstrlen(Name+1) 得到的实际结果对于 Unicode 字符来说比 ANSI 字符要小一半，而且它也不能从这个字符串的第二个字符起开始计算，也许你的本意是 lstrlen(Name) + 1（OK，我承认，我也犯了同样的错误！）。不论如何，在 Unicode 模式下，所有的字符都是2个字节大小，我们需要处理这个问题。微软的文档令人惊讶地对此保持缄默：REG\_SZ 的值究竟是以字节计算还是以字符计算呢？我们假设它指的是以字节为单位计算，你需要对你的代码做一些修改来计算这个字符串所含有的字节大小。

## 10、CString 的效率

　　CString 的一个问题是它确实掩藏了一些低效率的东西。从另外一个方面讲，它也确实可以被实现得更加高效，你可能会说下面的代码：

CString s = SomeCString1;s += SomeCString2;s += SomeCString3;s += ",";s += SomeCString4;

比起下面的代码来，效率要低多了：

char s[1024];lstrcpy(s, SomeString1);lstrcat(s, SomeString2);lstrcat(s, SomeString 3);lstrcat(s, ",");lstrcat(s, SomeString4);

　 　总之，你可能会想，首先，它为 SomeCString1 分配一块内存，然后把 SomeCString1 复制到里面，然后发现它要做一个连接，则重新分配一块新的足够大的内存，大到能够放下当前的字符串加上SomeCString2，把内容复制到这块内存，然后把 SomeCString2 连接到后面，然后释放第一块内存，并把指针重新指向新内存。然后为每个字符串重复这个过程。把这 4 个字符串连接起来效率多低啊。事实上，在很多情况下根本就不需要复制源字符串（在 += 操作符左边的字符串）。

　　在 VC++6.0 中，Release 模式下，所有的 CString 中的缓存都是按预定义量子分配的。所谓量子，即确定为 64、128、256 或者 512 字节。这意味着除非字符串非常长，连接字符串的操作实际上就是 strcat 经过优化后的版本（因为它知道本地的字符串应该在什么地方结束，所以不需要寻找字符串的结尾；只需要把内存中的数据拷贝到指定的地方即可）加上重新计算字符串的长度。所以它的执行效率和纯 C 的代码是一样的，但是它更容易写、更容易维护和更容易理解。

　　如果你还是不能确定究竟发生了怎样的过程，请看看 CString 的源代码，strcore.cpp，在你 vc98的安装目录的 mfc\src 子目录中。看看 ConcatInPlace 方法，它被在所有的 += 操作符中调用。

啊哈！难道 CString 真的这么"高效"吗？比如，如果我创建

CString cat("Mew!");

　　然后我并不是得到了一个高效的、精简的5个字节大小的缓冲区（4个字符加一个结束字符），系统将给我分配64个字节，而其中59个字节都被浪费了。

　　如果你也是这么想的话，那么就请准备好接受再教育吧。可能在某个地方某个人给你讲过尽量使用少的空间是件好事情。不错，这种说法的确正确，但是他忽略了事实中一个很重要的方面。

　　如果你编写的是运行在16K EPROMs下的嵌入式程序的话，你有理由尽量少使用空间，在这种环境下，它能使你的程序更健壮。但是在 500MHz, 256MB的机器上写 Windows 程序，如果你还是这么做，它只会比你认为的“低效”的代码运行得更糟。

　　举例来说。字符串的大小被认为是影响效率的首要因素，使字符串尽可能小可以提高效率，反之则降低效率，这是大家一贯的想法。但是这种想法是不对的，精确的内存分配的后果要在程序运行了好几个小时后才能体现得出来，那时，程序的堆中将充满小片的内存，它们太小以至于不能用来做任何事，但是他们增加了你程序的内存用量，增加了内存页面交换的次数，当页面交换的次数增加到系统能够忍受的上限，系统则会为你的程序分配更多的页面，直到你的程序占用了所有的可用内存。由此可见，虽然内存碎片是决定效率的次要因素，但正是这些因素实际控制了系统的行为，最终，它损害了系统的可靠性，这是令人无法接受的。

　　记住，在 debug 模式下，内存往往是精确分配的，这是为了更好的排错。

　 　假设你的应用程序通常需要连续工作好几个月。比如，我常打开 VC++，Word，PowerPoint，Frontpage，Outlook Express，Forté Agent，Internet Explorer和其它的一些程序，而且通常不关闭它们。我曾经夜以继日地连续用 PowerPoint 工作了好几天（反之，如果你不幸不得不使用像 Adobe FrameMaker 这样的程序的话，你将会体会到可靠性的重要；这个程序机会每天都要崩溃4~6次，每次都是因为用完了所有的空间并填满我所有的交换页面）。所以精确内存分配是不可取的，它会危及到系统的可靠性，并引起应用程序崩溃。

　　按量子的倍数为字符串分配内存，内存分配器就可以回收用过的内存块，通常这些回收的内存块马上就可以被其它的 CString 对象重新用到，这样就可以保证碎片最少。分配器的功能加强了，应用程序用到的内存就能尽可能保持最小，这样的程序就可以运行几个星期或几个月而不出现问题。

　　题外话：很多年以前，我们在 CMU 写一个交互式系统的时候，一些对内存分配器的研究显示出它往往产生很多内存碎片。Jim Mitchell，现在他在 Sun Microsystems 工作，那时侯他创造了一种内存分配器，它保留了一个内存分配状况的运行时统计表，这种技术和当时的主流分配器所用的技术都不同，且较为领先。当一个内存块需要被分割得比某一个值小的话，他并不分割它，因此可以避免产生太多小到什么事都干不了的内存碎片。事实上他在内存分配器中使用了一个浮动指针，他认为：与其让指令做长时间的存取内存操作，还不如简单的忽略那些太小的内存块而只做一些浮动指针的操作。（His observation was that the long-term saving in instructions by not having to ignore unusable small storage chunks far and away exceeded the additional cost of doing a few floating point operations on an allocation operation.）他是对的。

　　永远不要认为所谓的“最优化”是建立在每一行代码都高速且节省内存的基础上的，事实上，高速且节省内存应该是在一个应用程序的整体水平上考虑的。在软件的整体水平上，只使用最小内存的字符串分配策略可能是最糟糕的一种方法。

　　如果你认为优化是你在每一行代码上做的那些努力的话，你应该想一想：在每一行代码中做的优化很少能真正起作用。你可以看我的另一篇关于优化问题的文章《Your Worst Enemy for some thought-provoking ideas》。

　　记住，+= 运算符只是一种特例，如果你写成下面这样：

CString s = SomeCString1 + SomeCString2 + SomeCString3 + "," + SomeCString4;

则每一个 + 的应用会造成一个新的字符串被创建和一次复制操作。

总结

以上是使用 CString 的一些技巧。我每天写程序的时候都会用到这些。CString 并不是一种很难使用的类，但是 MFC 没有很明显的指出这些特征，需要你自己去探索、去发现。

# 15.string字符串与char[]字符数组相互转换

Char 转化为string

使用拷贝构造函数创建string对象：

string str = ch;

或者

string str;

str = ch;

例如：

Char ch[5]=“abcd”;

String str=ch;

String 转换为char[]

String str=”abcd”;

Char ch[20]; //数值要比strlen()大1.

strcpy\_s(ch,str.c\_str()); //使用c\_str()函数。有的版本用strcpy().

# 遍历文件夹

如何将生成的数据放在一个文件夹里？？？

最简单的办法，在当前项目的路径下手动新建一个文件夹，如TempData,

其他路径按：“c:/user/docments/TempData” 左斜杠设置即可；

然后保存数据是加一路径，如outfile(“TempData/data1;ios::out”);

包含头文件 #include <io.h>

　　用到数据结构\_finddata\_t，文件信息结构体的指针。

　　struct\_finddata\_t

　　{

　　unsigned attrib;//文件属性

　　time\_t time\_create;//文件创建时间

　　time\_t time\_access;//文件上一次访问时间

　　time\_t time\_write;//文件上一次修改时间

　　\_fsize\_t size;//文件字节数

　　char name[\_MAX\_FNAME];//文件名

　　};

　　文件属性是无符号整数，取值为相应的宏：\_A\_ARCH(存档)，\_A\_SUBDIR(文件夹)，\_A\_HIDDEN(隐藏)，\_A\_SYSTEM(系统)，\_A\_NORMAL(正常)，\_A\_RDONLY(只读)。容易看出，通过这个结构体，我们可以得到关于该文件的很多信息。结合以下函数，我们可以将文件信息存储到这个结构体中：

　　//按FileName命名规则匹配当前目录第一个文件

　　\_findfirst(\_In\_ constchar\* FileName,\_Out\_ struct\_finddata64i32\_t \*\_FindData);

　　//按FileName命名规则匹配当前目录下一个文件

　　\_findnext(\_In\_ intptr\_t \_FindHandle,\_Out\_ struct\_finddata64i32\_t \*\_FindData);

　　//关闭\_findfirst返回的文件句柄

　　\_findclose(\_In\_ intptr\_t \_FindHandle);

　　\_findfirst 函数返回的是匹配到文件的句柄，数据类型为long。遍历过程可以指定文件类型。

实例：

int \_tmain(int argc, \_TCHAR\* argv[])

{

bool TraverseFiles(string path, int&file\_num);

void DfsFolder(string path, int layer);

//遍历单个目录

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*/遍历单个目录\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

int file\_num = 0;

if (!TraverseFiles("C:/Users/Administrator/Desktop", file\_num))//此处路径连接符只能用/，根盘符大小写都行

cout << "traverse files failed" << endl;

cout << endl;

cout << "-------------------n" << "file number: " << file\_num << endl;

//递归遍历文件夹

cout << "\*\*\*\*\*\*\*\*\*\*\*\*递归遍历文件夹\*\*\*\*\*\*\*\*\*\*\*" << endl;

DfsFolder("C:/Users/Administrator/Desktop/example", 0);

system("pause");

return 0;

}

bool TraverseFiles(string path, int&file\_num)

{

\_finddata\_t file\_info;

string current\_path = path + "/\*.\*";//可以定义后面的后缀为\*.exe，\*.txt等来查找特定后缀的文件，\*.\*是通配符，匹配所有类型,路径连接符最好是左斜杠/，可跨平台

//打开文件查找句柄

int handle = \_findfirst(current\_path.c\_str(), &file\_info);

//返回值为-1则查找失败

if (-1 == handle)

return false;

do

{

//判断是否子目录

string attribute;

if (file\_info.attrib == \_A\_SUBDIR)//是目录

attribute = "dir";

else

attribute = "file";

//输出文件信息并计数,文件名(带后缀)、文件最后修改时间、文件字节数(文件夹显示0)、文件是否目录

cout << file\_info.name << ' ' << file\_info.time\_write << ' ' << file\_info.size << ' ' << attribute << endl;//获得的最后修改时间是time\_t格式的长整型，需要用其他方法转成正常时间显示

file\_num++;

} while (!\_findnext(handle, &file\_info));//返回0则遍历完

//关闭文件句柄

\_findclose(handle);

return true;

}

//深度优先递归遍历当前目录下文件夹和文件及子文件夹和文件

void DfsFolder(string path, int layer)

{

\_finddata\_t file\_info;

string current\_path = path + "/\*.\*";//也可以用/\*来匹配所有

int handle = \_findfirst(current\_path.c\_str(), &file\_info);

//返回值为-1则查找失败

if (-1 == handle)

{

cout << "cannot match the path" << endl;

return;

}

int i;

do

{

//判断是否子目录

if (file\_info.attrib == \_A\_SUBDIR)

{

//递归遍历子目录

//打印记号反映出深度层次

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

cout << "--";

cout << file\_info.name << endl;

int layer\_tmp = layer;

if (strcmp(file\_info.name, "..") != 0 && strcmp(file\_info.name, ".") != 0)//.是当前目录，..是上层目录，必须排除掉这两种情况

DfsFolder(path + '/' + file\_info.name, layer\_tmp + 1);//再windows下可以用转义分隔符，不推荐

}

else

{

//打印记号反映出深度层次

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

cout << "--";

cout << file\_info.name << endl;

}

} while (!\_findnext(handle, &file\_info));//返回0则遍历完

//关闭文件句柄

\_findclose(handle);

}

# 17.标准C++中的string类的用法总结

要想使用标准C++中string类，必须要包含

#include <string>// 注意是<string>，不是<string.h>，带.h的是C语言中的头文件

using string;

using wstring;

或

using namespace std;

下面你就可以使用string/wstring了，它们两分别对应着char和wchar\_t。

string和wstring的用法是一样的，以下只用string作介绍：

## 1.string类的构造函数：

string(const char \*s); //用c字符串s初始化

string(int n,char c); //用n个字符c初始化

此外，string类还支持默认构造函数和复制构造函数，如string s1；string s2="hello"；都是正确的写法。当构造的string太长而无法表达时会抛出length\_error异常 ；

## 2.string类的字符操作：

const char &operator[](int n)const;

const char &at(int n)const;

char &operator[](int n);

char &at(int n);

operator[]和at()均返回当前字符串中第n个字符的位置，但at函数提供范围检查，当越界时会抛出out\_of\_range异常，下标运算符[]不提供检查访问。

const char \*data()const;//返回一个非null终止的c字符数组

const char \*c\_str()const;//返回一个以null终止的c字符串

int copy(char \*s, int n, int pos = 0) const;//把当前串中以pos开始的n个字符拷贝到以s为起始位置的字符数组中，返回实际拷贝的数目

## 3.string的特性描述:

int capacity()const; //返回当前容量（即string中不必增加内存即可存放的元素个数）

int max\_size()const; //返回string对象中可存放的最大字符串的长度

int size()const; //返回当前字符串的大小

int length()const; //返回当前字符串的长度

bool empty()const; //当前字符串是否为空

void resize(int len,char c);//把字符串当前大小置为len，并用字符c填充不足的部分

## 4.string类的输入输出操作:

string类重载运算符operator>>用于输入，同样重载运算符operator<<用于输出操作。

函数getline(istream &in,string &s);用于从输入流in中读取字符串到s中，以换行符'\n'分开。

string的赋值：

string &operator=(const string &s);//把字符串s赋给当前字符串

string &assign(const char \*s);//用c类型字符串s赋值

string &assign(const char \*s,int n);//用c字符串s开始的n个字符赋值

string &assign(const string &s);//把字符串s赋给当前字符串

string &assign(int n,char c);//用n个字符c赋值给当前字符串

string &assign(const string &s,int start,int n);//把字符串s中从start开始的n个字符赋给当前字符串

string &assign(const\_iterator first,const\_itertor last);//把first和last迭代器之间的部分赋给字符串

## 5.string的连接：

string &operator+=(const string &s);//把字符串s连接到当前字符串的结尾

string &append(const char \*s); //把c类型字符串s连接到当前字符串结尾

string &append(const char \*s,int n);//把c类型字符串s的前n个字符连接到当前字符串结尾

string &append(const string &s); //同operator+=()

string &append(const string &s,int pos,int n);//把字符串s中从pos开始的n个字符连接到当前字符串的结尾

string &append(int n,char c); //在当前字符串结尾添加n个字符c

string &append(const\_iterator first,const\_iterator last);//把迭代器first和last之间的部分连接到当前字符串的结尾

## 6.string的比较：

bool operator==(const string &s1,const string &s2)const;//比较两个字符串是否相等

运算符">","<",">=","<=","!="均被重载用于字符串的比较；

int compare(const string &s) const;//比较当前字符串和s的大小

int compare(int pos, int n,const string &s)const;//比较当前字符串从pos开始的n个字符组成的字符串与s的大小

int compare(int pos, int n,const string &s,int pos2,int n2)const;//比较当前字符串从pos开始的n个字符组成的字符串与s中

　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　//pos2开始的n2个字符组成的字符串的大小

int compare(const char \*s) const;

int compare(int pos, int n,const char \*s) const;

int compare(int pos, int n,const char \*s, int pos2) const;

compare函数在>时返回1，<时返回-1，==时返回0

## 7.string的子串：

string substr(int pos = 0,int n = npos) const;//返回pos开始的n个字符组成的字符串

## 8.string的交换：

void swap(string &s2); //交换当前字符串与s2的值

## 9.string类的查找函数：

**没有找到返回 -1**

int find(char c, int pos = 0) const;//从pos开始查找字符c在当前字符串的位置

int find(const char \*s, int pos = 0) const;//从pos开始查找字符串s在当前串中的位置

int find(const char \*s, int pos, int n) const;//从pos开始查找字符串s中前n个字符在当前串中的位置

int find(const string &s, int pos = 0) const;//从pos开始查找字符串s在当前串中的位置

//查找成功时返回所在位置，失败返回string::npos的值

int rfind(char c, int pos = npos) const;//从pos开始从后向前查找字符c在当前串中的位置

int rfind(const char \*s, int pos = npos) const;

int rfind(const char \*s, int pos, int n = npos) const;

int rfind(const string &s,int pos = npos) const;

//从pos开始从后向前查找字符串s中前n个字符组成的字符串在当前串中的位置，成功返回所在位置，失败时返回string::npos的值

int find\_first\_of(char c, int pos = 0) const;//从pos开始查找字符c第一次出现的位置

int find\_first\_of(const char \*s, int pos = 0) const;

int find\_first\_of(const char \*s, int pos, int n) const;

int find\_first\_of(const string &s,int pos = 0) const;

//从pos开始查找当前串中第一个在s的前n个字符组成的数组里的字符的位置。查找失败返回string::npos

int find\_first\_not\_of(char c, int pos = 0) const;

int find\_first\_not\_of(const char \*s, int pos = 0) const;

int find\_first\_not\_of(const char \*s, int pos,int n) const;

int find\_first\_not\_of(const string &s,int pos = 0) const;

//从当前串中查找第一个不在串s中的字符出现的位置，失败返回string::npos

int find\_last\_of(char c, int pos = npos) const;

int find\_last\_of(const char \*s, int pos = npos) const;

int find\_last\_of(const char \*s, int pos, int n = npos) const;

int find\_last\_of(const string &s,int pos = npos) const;

int find\_last\_not\_of(char c, int pos = npos) const;

int find\_last\_not\_of(const char \*s, int pos = npos) const;

int find\_last\_not\_of(const char \*s, int pos, int n) const;

int find\_last\_not\_of(const string &s,int pos = npos) const;

//find\_last\_of和find\_last\_not\_of与find\_first\_of和find\_first\_not\_of相似，只不过是从后向前查找

## 10.string类的替换函数：

string &replace(int p0, int n0,const char \*s);//删除从p0开始的n0个字符，然后在p0处插入串s

string &replace(int p0, int n0,const char \*s, int n);//删除p0开始的n0个字符，然后在p0处插入字符串s的前n个字符

string &replace(int p0, int n0,const string &s);//删除从p0开始的n0个字符，然后在p0处插入串s

string &replace(int p0, int n0,const string &s, int pos, int n);//删除p0开始的n0个字符，然后在p0处插入串s中从pos开始的n个字符

string &replace(int p0, int n0,int n, char c);//删除p0开始的n0个字符，然后在p0处插入n个字符c

string &replace(iterator first0, iterator last0,const char \*s);//把[first0，last0）之间的部分替换为字符串s

string &replace(iterator first0, iterator last0,const char \*s, int n);//把[first0，last0）之间的部分替换为s的前n个字符

string &replace(iterator first0, iterator last0,const string &s);//把[first0，last0）之间的部分替换为串s

string &replace(iterator first0, iterator last0,int n, char c);//把[first0，last0）之间的部分替换为n个字符c

string &replace(iterator first0, iterator last0,const\_iterator first, const\_iterator last);//把[first0，last0）之间的部分替换成[first，last）之间的字符串

## 11.string类的插入函数：

string &insert(int p0, const char \*s);

string &insert(int p0, const char \*s, int n);

string &insert(int p0,const string &s);

string &insert(int p0,const string &s, int pos, int n);

//前4个函数在p0位置插入字符串s中pos开始的前n个字符

string &insert(int p0, int n, char c);//此函数在p0处插入n个字符c

iterator insert(iterator it, char c);//在it处插入字符c，返回插入后迭代器的位置

void insert(iterator it, const\_iterator first, const\_iterator last);//在it处插入[first，last）之间的字符

void insert(iterator it, int n, char c);//在it处插入n个字符c

## 12.string类的删除函数

iterator erase(iterator first, iterator last);//删除[first，last）之间的所有字符，返回删除后迭代器的位置

iterator erase(iterator it);//删除it指向的字符，返回删除后迭代器的位置

string &erase(int pos = 0, int n = npos);//删除pos开始的n个字符，返回修改后的字符串

## 13.string类的迭代器处理：

string类提供了向前和向后遍历的迭代器iterator，迭代器提供了访问各个字符的语法，类似于指针操作，迭代器不检查范围。

用string::iterator或string::const\_iterator声明迭代器变量，const\_iterator不允许改变迭代的内容。常用迭代器函数有：

const\_iterator begin()const;

iterator begin(); //返回string的起始位置

const\_iterator end()const;

iterator end(); //返回string的最后一个字符后面的位置

const\_iterator rbegin()const;

iterator rbegin(); //返回string的最后一个字符的位置

const\_iterator rend()const;

iterator rend(); //返回string第一个字符位置的前面

rbegin和rend用于从后向前的迭代访问，通过设置迭代器string::reverse\_iterator,string::const\_reverse\_iterator实现

## 14.string 与char[]的大小确定函数

int \_tmain(int argc, \_TCHAR\* argv[])

{

cout << "--------------------\n";

string x = "123456";

char y[] = "12345";

int s2 = sizeof(y); //s2=6; 多一个'\0' char 使用sizeof()-1确定实际大小

cout << x << endl;

int s1 = strlen(y); //s1=5 实际长度

cout << "char[]=" << s1 << endl;

int s = x.length(); //string 字符串类型使用length()函数

cout << s << endl;

x.resize(10, 'a'); //重新设置字符串的大小，并将不足大小的位置使用字符a填充

cout << x << endl;

cout << x.length();

system("pause");

return 0;}

## 15.string 的resize()函数

resize函数用来改变string的大小，如果size小于来string的size大小，则截取前size个字符，如果大于，则用空格补充。

s.resize(20,'a');

填充函数！如果s字符串长度不足20，则不足部分用字符a填充。;

#include<iostream>

#include<string>

using namespace std;

int main(){

    string s = "ab";

    s.resize(20,'a');

    cout<<s.length()<<endl;

    cout<<s.capacity()<<endl;

    cout<<s<<endl;

    s.resize(12);

    cout<<s.length()<<endl;

    cout<<s<<endl;

}

输出结果为：

20

20

abaaaaaaaaaaaaaaaaaa

12

abaaaaaaaaaa

# 18.文件读写

void fun1()

{

ofstream outfile1("f1.dat");

ofstream outfile2("f2.dat");

int a[20];

int i;

for (i = 0; i < 20; i++)

{

cin >> a[i];

if (i < 10)

outfile1 << a[i] << " ";

else

outfile2 << a[i] << " ";

}

outfile1.close();

outfile2.close();

}

void fun2()

{

ifstream infile1("f1.dat",ios::in);

ofstream outfile2("f2.dat",ios::app);

char ch;

while(infile1.get(ch)) //从f1.dat文件读取数据

{

outfile2.put(ch) ; //将数据输出到f2.dat

}

infile1.close();

outfile2.close();

}

void fun3()

{

ifstream infile1("f2.dat", ios::in);

ofstream outfile2("f1.dat", ios::app);

string str;

while (getline(infile1,str)) //从f2.dat文件读取数据

{

outfile2 << str<<endl; //整行整行输出

}

infile1.close();

outfile2.close();

}

## 推荐方法

**void fun4()**

{

ifstream infile("f1.dat");

ofstream outfile("f4.dat");

string str;

while (getline(infile, str)) //**完全相同复制:截取**

{

outfile << str << endl;

}

infile.close();

outfile.close();

}

void fun5()

{

ifstream infile("f1.dat");

ofstream outfile("f5.dat");

string str; //字符串存放

while (!infile.eof()) // //值得注意：这个判断会多输出一空行

{

getline(infile, str);

outfile << str << endl;

}

infile.close();

outfile.close();

}

void fun6()

{

ifstream infile("f1.dat");

ofstream outfile("f5.dat");

char str[100]; //字符数组存放

while (!infile.eof()) // //值得注意：这个判断会多输出一空行

{

infile.getline(str, 100);

outfile << str << endl;

}

infile.close();

outfile.close();

}

## Infile流对象直接读取 VS getline(infile,str)

算例文档test.txt如下：

1 2 33.0 55 6.66 888 lsl 5

5 3 66.0 78 555 987 kjh 4

代码如下：

int \_tmain(int argc, \_TCHAR\* argv[])

{

ifstream infile("test.txt", ios::in);

if (!infile)

{

cerr << "open file error" << endl;

}

string str;

getline(infile, str); //按字符串形式读取第一行

cout << str << endl;

double a, b, c, d, e, f, h;

string name;

infile >> a >> b >> c >> d >> e >> f >> name >> h; //按数据形式自动分空格读取第二行，很方便

cout << a << " " << b << " " << c << " " << d << " " << e << " " << f << " " << name << " " << h << endl;

infile.close();

system("pause");

return 0;

}

要点：getline(infile,str)函数按字符串形式**读取一行**，读取完后要根据字符长度截取相应数据并转化格式；

infile >> a >> b >> c >> d >> e >> f >> name >> h;

//按数据形式自动分空格读取行并储存为相应变量，如显示屏getchar()函数类似，自动读取行并保存，很方便。

**总结：在每行的数据形式有规则情形，且只需要读取一次，可优先使用infile函数读取！**

**/\***

**//-------------------**

**string str = " ";**

**int num = 0;**

**double a1 = 0.0; double b1 = 0.0;**

**ifstream intext("smenergy.txt", ios::in);**

**ofstream outtext1("smenergy1.txt", ios::out);**

**ofstream outtext2("smenergy2.txt", ios::out);**

**while (!intext.eof())**

**{**

**num++;**

**intext >> a1 >> b1;**

**if (fmodf(num, 2))**

**{**

**outtext1 << setw(8) << std::setprecision(2) << std::setiosflags(ios::showpoint | ios::fixed) << a1**

**<< setw(10) << std::setprecision(5) << std::setiosflags(ios::showpoint | ios::fixed) << b1 << endl;**

**}**

**else**

**outtext2 << setw(8) << std::setprecision(2) << std::setiosflags(ios::showpoint | ios::fixed) << a1**

**<< setw(10) << std::setprecision(5) << std::setiosflags(ios::showpoint | ios::fixed) << b1 << endl;**

**}**

**//-------------------**

**exit(1);**

**\*/**

**在数据无规则情形下，或者需要对该行数据重复读取时，可用getline函数将行保存为字符串变量，从而寄放在存储中，方便以后使用；**

**本质：infile就是一个自动换行读取的函数！**

# 19.unsinged ,long ,short

加入unsigned 关键字 表名**不含负数**。

比如 int 范围-32768~32767

而 unsigned int 范围 0~65535

[c++ 中关于int，unsigned int , short的关系与应用](http://www.cppblog.com/xyjzsh/archive/2010/10/20/130554.html)

int类型比较特殊，具体的字节数同机器字长和编译器有关。如果要保证移植性，尽量用\_\_int16 \_\_int32 \_\_int64吧  
\_\_int16、\_\_int32这种数据类型在所有平台下都分配相同的字节。所以在移植上不存在问题。  
所谓的不可移植是指：在一个平台上编写的代码无法拿到另一个平台上运行时，不能达到期望的运行结果。  
例如：在32为平台上（所谓32位平台是指通用寄存器的数据宽度是32）编写代码，int 类型分配4个字节，而在16位平台是则分配2个字节，那么在16位上编译出来的exe，  
其中是为int分配2字节，而在32位平台上运行时，会按照4个字节来解析，显然会出错误的！！  
  
而对于非int行，目前为止，所有的类型分配的字节数都是兼容的，即不同平台对于同一个类型分配相同的字节数！！

## 建议：

**在代码中尽量避免使用int类型，根据不同的需要可以用short,long,unsigned int 等代替。**  
  
下面是各个类型一览表【转】

64位指的是cpu通用寄存器的数据宽度是64位的。

|  |  |  |  |
| --- | --- | --- | --- |
| **数据类型名称** | **字节数** | **别名** | **取值范围** |
| **int** | \* | signed,signed int | 由[操作系统](http://dev.21tx.com/os/)决定，即与操作系统的＂字长＂有关 |
| **unsigned int** | \* | unsigned | 由操作系统决定，即与操作系统的＂字长＂有关 |
| **\_\_int8** | 1 | char,signed char | –128 到 127 |
| **\_\_int16** | 2 | short,short int,signed short int | –32,768 到 32,767 |
| **\_\_int32** | 4 | signed,signed int | –2,147,483,648 到 2,147,483,647 |
| **\_\_int64** | 8 | 无 | –9,223,372,036,854,775,808 到 9,223,372,036,854,775,807 |
| **bool** | 1 | 无 | false 或 true |
| **char** | 1 | signed char | –128 到 127 |
| **unsigned char** | 1 | 无 | 0 到 255 |
| **short** | 2 | short int,signed short int | –32,768 到 32,767 |
| **unsigned short** | 2 | unsigned short int | 0 到 65,535 |
| **long** | 4 | long int,signed long int | –2,147,483,648 到 2,147,483,647 |
| **long long** | 8 | none (but equivalent to \_\_int64) | –9,223,372,036,854,775,808 到 9,223,372,036,854,775,807 |
| **unsigned long** | 4 | unsigned long int | 0 到 4,294,967,295 |
| **enum** | \* | 无 | 由操作系统决定，即与操作系统的＂字长＂有关 |
| **float** | 4 | 无 | 3.4E +/- 38 (7 digits) |
| **double** | 8 | 无 | 1.7E +/- 308 (15 digits) |
| **long double** | 8 | 无 | 1.7E +/- 308 (15 digits) |
| **wchar\_t** | 2 | \_\_wchar\_t | 0 到 65,535 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型标识符 | 类型说明 | 长度 （字节） | 范围 | 备注 |
| **char** | 字符型 | 1 | -128 ~ 127 | -27 ~ (27 -1) |
| **unsigned char** | 无符字符型 | 1 | 0 ~ 255 | 0 ~ (28 -1) |
| **short int** | 短整型 | 2 | -32768 ~ 32767 | 2-15 ~ (215 - 1) |
| **unsigned short int** | 无符短整型 | 2 | 0 ~ 65535 | 0 ~ (216 - 1) |
| **int** | 整型 | 4 | -2147483648 ~ 2147483647 | -231 ~ (231 - 1) |
| **unsigned int** | 无符整型 | 4 | 0 ~ 4294967295 | 0 ~ (232-1) |
| **float** | 实型（单精度） | 4 | 1.18\*10-38 ~ 3.40\*1038 | 7位有效位 |
| **double** | 实型（双精度） | 8 | 2.23\*10-308~ 1.79\*10308 | 15位有效位 |
| **long double** | 实型（长双精度） | 10 | 3.37\*10-4932 ~ 1.18\*104932 | 19位有效位 |

# 20.auto类型变量——根据初始值推断真实的数据类型。

【为什么auto】

auto x=7; 对

auto x; x=7; 错

对于模板类型特别有用，如

template <typename T，typename U>

{

auto x=T\*U;

...

}  
 在开发实践中，有时候我们并不能非常容易地确定一个变量应该具有的数据类型。比如，将某个复杂[表达式](http://www.07net01.com/tags-%E8%A1%A8%E8%BE%BE%E5%BC%8F-0.html)作为初始值赋值给一个新定义的变量时，我们往往很难确定这个表达式的数据类型，从而无法确定变量应有的数据类型。为了解决这个问题，C++11为我们提供了auto关键字，使用它作为某个变量定义的数据类型，[编译器](http://www.07net01.com/tags-%E7%BC%96%E8%AF%91%E5%99%A8-0.html)会根据这个变量的初始值，自动推断出这个变量合理的数据类型而无需我们人为指定。

【auto概念详解】

auto关键字——根据初始值推断真实的数据类型。

auto[存储](http://www.07net01.com/storage_networking/)类型说明

auto存储类型说明符声明了一个自动变量，auto对象和变量被存储在栈中，它的生命周期仅存在于它的声明所在的块(block)中，即一个只在块运行时有效的变量。一个auto变量仅在声明它的块内是可见的。auto对象和变量对外部模块都是不可见的。auto变量的声明能包含初始化。因为有auto存储类型的变量并不自动的初始化，应该在声明时显式的初始化它们，或在同一个块内赋给它们初始值。未初始化的auto变量的值是未定义的。

【auto代码使用示例】

auto x = 7; // 使用整数7对变量x进行初始化，x被推断为int类型 auto y = 1.982; // 使用浮点数1.982对变量y进行初始化，y被推断为double类型。虽然auto关键字会根据初始值自动推断变量的数据类型，但是，它的使用并不需要花费额外的编译时间。

实际上，可以把auto关键字看成是一个变量定义中的数据类型占位符，它占据了原来应该是具体数据类型的位置。而在编译的时候，编译器会根据这个变量的初始值，推断出这个变量应有的具体数据类型，然后替换掉auto关键字，就成为一个普通的带有具体数据类型的变量定义了。用auto关键字定义变量的形式跟一般的定义变量的形式并无二异，唯一的差别之处在于，用auto关键字定义变量时，变量必须有初始值：

auto 变量名 = 初始值表达式; // 赋值形式 // 或 auto 变量名{初始值表达式}; // 初始化列表形式这样，这个初始值表达式计算结果的数据类型将被编译器推断为变量的数据类型。

template <typename T> // 数据类型vector<T>之后的“&”符号，表示其后所定义的变量是一个引用 // 引用是C++中一种访问数据的特殊方式，在稍后的7.1小节中我们将详细介绍 void printall(const vector<T>& v) { // 根据v.begin()的返回值类型自动推断变量it的数据类型 for (auto it = v.begin(); it != v.end(); ++it) cout << \*it << endl; }为了表示同样的意义，如果没有auto关键字帮忙，我们不得不写成下面这种繁琐的形式：

template <typename T> void printall(const vector<T>& v) { for (typename vector<T>::const\_iterator it = v.begin(); it != v.end(); ++it) cout << \*it << endl; }除了简化代码之外，auto关键字有时候甚至能够帮助我们完成一些在C++11之前不可能完成的任务，成为一种必需。比如，在模板函数中，当一个变量的数据类型依赖于模板参数时，如果不使用auto关键字，将根本无法确定变量的数据类型，因为我们根本无法提前预知用户使用何种数据类型作为模板参数来调用这个模板函数，从而也就无法确定这个变量的数据类型。但是使用auto关键字之后，一切难题都将迎刃而解。例如：

template <typename T,typename U> void mul(const T& t,const U& u) { // ... // 用auto关键字做数据类型，编译器将根据u和t的实际数据类型， // 自动推断变量tmp的数据类型 auto tmp = t\*u; // ... }在这里，变量tmp的数据类型应该与模板参数T和U相乘结果的数据类型相同，也就是依赖于T和U的数据类型。对于[程序](http://www.07net01.com/tags-%E7%A8%8B%E5%BA%8F-0.html)员来说，在编写这个模板函数的时候，模板参数T和U的类型尚未确定，这样变量tmp的类型也就无法确定。所以，我们用auto关键字作为占位符，占据数据类型的位置，而真正的数据类型，则留待编译器在最终编译的时候，根据具体给定的模板参数T和U的类型而推断得到。这样，就把一件原来不可能的事情变成了可能。  
【auto注意事项】  
什么时候不应该用auto:  
1.需要明确的告诉阅读者类型的时候。(代码可读性问题)  
2.要在多个编译器下面编译的时候。(并非每个编译器都支持auto)  
3.当你要定义string而用const char\*初始化的时候（以及类似的情况）。

4.过去的代码中已经使用了auto，但是并非自动推断类型的意义的时候如：auto int a = 100;如果开启自动推断那么这是错误的语法。

【另】

这样的代码是不能通过编译的：

auto a;

因为auto关键字要求必须有初始值。error C3531: “a”: 类型包含“auto”的符号必须具有初始值设定项。

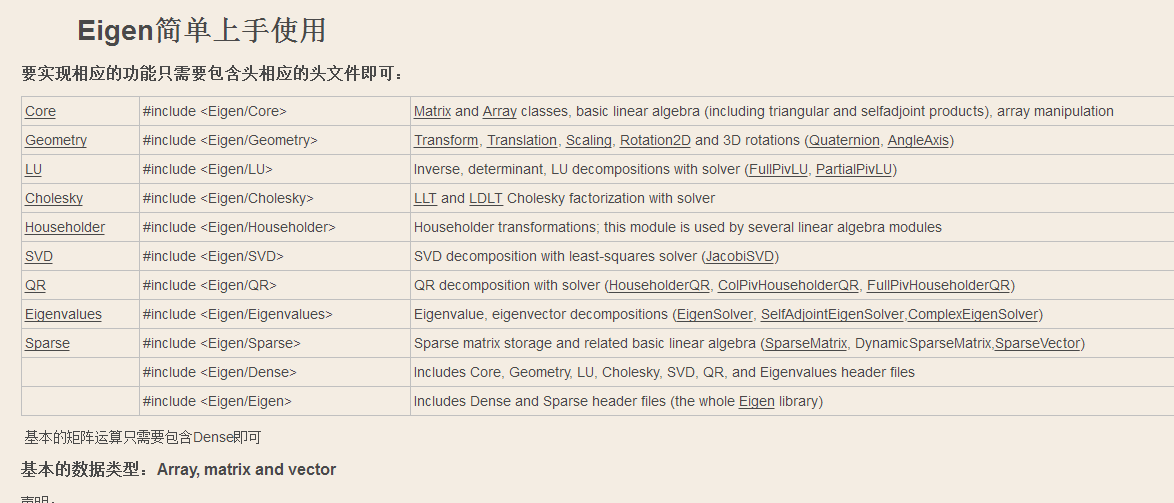
# Eigen3矩阵模块的使用

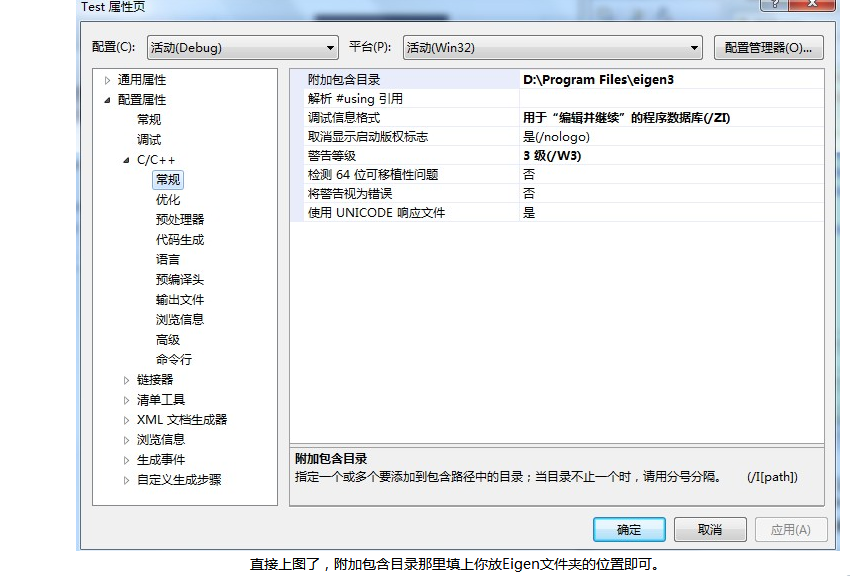
**Eigen是C++中可以用来调用并进行矩阵计算的一个库，里面封装了一些类。**

参考1：<http://blog.csdn.net/abcjennifer/article/details/7781936>

参考2：<http://www.cnblogs.com/goingupeveryday/p/5699053.html>

参考3：<http://blog.csdn.net/allyli0022/article/details/52347900>





操作对象：Array, matrix and vector（数组，矩阵和向量）

## 常用模块头文件:

#include <Eigen/Core>

#include <Eigen/Geometry>

#include <Eigen/LU>

#include <Eigen/Cholesky>

#include <Eigen/Householder>

#include <Eigen/SVD>

#include <Eigen/QR>

#include <Eigen/Eigenvalues>

#include <Eigen/Sparse>

#include <Eigen/Dense>

#include <Eigen/Eigen>

using namespace Eigen;

using namespace Eigen::internal;

using namespace Eigen::Architecture;

（红色为必要！）

**D:\Documents\Visual Studio 2013\eigen3**

附：QR分解

//QR分解的三种方法：1、FullPivHouseholderQR 2、ColPivHouseholderQR 3、HouseholderQR

//以下为HouseholderQR分解方法演示

Matrix3d A;

A << 1, 1, 1,

2, -1, -1,

2, -4, 5;

HouseholderQR<Matrix3d> qr;

qr.compute(A);

MatrixXd R = qr.matrixQR().triangularView<Upper>();

MatrixXd Q = qr.householderQ();

cout << "QR2(): HouseholderQR---------------------------------------------" << endl;

cout << "A " << endl << A << endl << endl;

cout << "qr.matrixQR()" << endl << qr.matrixQR() << endl << endl;

cout << "R" << endl << R << endl << endl;

cout << "Q " << endl << Q << endl << endl;

cout << "Q\*R" << endl << Q\*R << endl << endl;

## MatrixXd,VectorXd做参数

特点：

MatrixXd或者VectorXd做参数时，跟数组不同

1. **Void型函数的参数值运算及改变不会传递到矩阵本身出来**
2. 不能使用const定义，如

void funmat(const MatrixXd m,const VectorXd v)则会出错

1. 解决方法

使用MatrixXd类型的具有返回值的函数

MatrixXd funmat(MatrixXd m,VectorXd v);

int \_tmain(int argc, \_TCHAR\* argv[])

{

Matrix3d mat;

mat << 1, 2, 3,

4, 5, 6,

7, 8, 9;

Vector3d v;

v << 3, 2, 1;

cout << "v=\n" << v << endl;

cout << "mat=\n" << mat << endl;

void funmat(MatrixXd m,VectorXd v);

funmat(mat,v);

cout << "mat=\n" << mat << endl;

cout << "v=\n" << v << endl;

system("pause");

return 0;

}

void funmat(MatrixXd m,VectorXd v)

{

m(1, 1) = 20;

v(1) = 10;

cout << "m=\n" << m << endl;

cout << "v=\n" << v << endl;

}

# 22.rand()函数及其用法

/\*

//----rand()与srand()函数：头文件为： #include"stdlib.h" , #include<time.h> //产生随机种子

\*\*\*1）int rand()函数返回0~RAND\_MAX间的一个随机整数；void srand()随机数种子函数，若不使用，默认为1，即每次rand()产生的随机数相同（便于调试）

\*\*\*若想利用 int rand()产生时刻不同的随机数，则可以使用srand(time(NULL))，从而随机数互不相同；

\*\*\*2）更改rand()随机数产生的范围：

\*\*\*使用rand() % n 可产生范围 0~n - 1的随机数 ，如：rand() % 11 ,产生0~10的随机整数

\*\*\*使用n + rand() % (m - n + 1); 范围 n~m ,r如: 1+rand() % (10-1+1),产生1~10的随机整数

\*\*\*3）利用rand()产生浮点数（1位小数） 0~1（不含1） ：(rand()%10)/(10.0)

\*\*\*2位小数0~1：(rand()%10)/(10.0) +(rand()%10)/(100.0)

\*\*\*3位小数0~1：(rand()%10)/(10.0) +(rand()%10)/(100.0)+(rand()%11)/(1000.0);

\*\*\*4位小数0~1：(rand()%10)/(10.0) +(rand()%10)/(100.0)+(rand()%11)/(1000.0)+(rand()%10)/(10000.0);依次类推

\*\*\*4）

\*\*\*

\*/

/\*

srand(time(NULL));

cout << "RAND\_MAX=" << RAND\_MAX << endl;

for (int i = 0; i < 20; i++)

{

double b = (rand() % 10) / (10.0) +(rand() % 10) / (100.0) +(rand() % 10) / (1000.0) +(rand() % 10) / (10000.0);

cout << "b=" << b << endl;

}

# 23 CArray与Vector删除元素的操作

## Vector

函数为erase(pos);删除指定位置的元素，之后vector的大小（size）减少。

vector<int>a;

for(int i = 0; i < 10; i++) a.push\_back(i);

cout << a.size() << endl;

for (int i = 0; i < 10; i++) cout << a.at(i) << " "; cout << endl;

a.erase(a.begin()); **//删除第一个元素**

cout << a.size() << endl;

for (int i = 0; i < a.size(); i++) cout << a.at(i) << " "; cout << endl;

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

vector <int> v1;

vector <int>::iterator Iter;

v1.push\_back(10);

v1.push\_back(20);

v1.push\_back(30);

v1.push\_back(40);

v1.push\_back(50);

cout << v1.size() << endl;

cout << "v1 =";

for (Iter = v1.begin(); Iter != v1.end(); Iter++)

cout << " " << \*Iter;

cout << endl;

v1.erase(v1.begin());//删除v1的第一个元素

cout << v1.size() << endl;

cout << "v1 =";

for (Iter = v1.begin(); Iter != v1.end(); Iter++)

cout << " " << \*Iter;

cout << endl;

//删除v1.begin() + 1和v1.begin() + 2两个元素

v1.erase(v1.begin() + 1, v1.begin() + 3);

cout << v1.size() << endl;

cout << "v1 =";

for (Iter = v1.begin(); Iter != v1.end(); Iter++)

cout << " " << \*Iter;

cout << endl;

for (Iter = v1.begin(); Iter != v1.end(); Iter++)

{

if (\*Iter == 50) **//删除指定大小的元素**

{

v1.erase(Iter);

Iter = v1.begin(); //当erase后，旧的容器会被重新整理成一个新的容器

}

}

cout << v1.size() << endl;

cout << "v1 =";

for (Iter = v1.begin(); Iter != v1.end(); Iter++)

cout << " " << \*Iter;

cout << endl;

exit(1);

## CArray

函数为Removeat(),删除后长度即减少1.

# 24 从N个元素中选取m个组合的实现

## 方法一：利用递归思想。

//从后往前选取，选定位置i后，再在前i-1个里面选取m-1个。

//如 1 2 3 4 5 中选取 3 个

//1、选取5后，再在前4个里面选取2个，而前4个里面选取2个又是一个子问题，递归即可。

//2、如果不包含5，直接选定4，那么再在前3个里面选取2个，而前三个里面选取2个又是一个子问题，递归即可。

//3、如果也不包含4，直接选取3，那么再在前2个里面选取2个，刚好只有两个。

//纵向看，1、2、3刚好是一个for循环，初值为5，终值为m

//横向看，该问题为一个前i-1个中选m-1的递归。

void Combination(int arr[], int nLen, int m, int out[], int outLen)

{

if(m == 0)

{

for (int j = 0; j < outLen; j++)

cout << out[j] << "\t";

cout << endl;

return;

}

for (int i = nLen; i >= m; --i) //从后往前依次选定一个

{

out[m-1] = arr[i-1]; //选定一个后

Combination(arr,i-1,m-1,out,outLen); // 从前i-1个里面选取m-1个进行递归

}

}

void PrintCombination(int arr[], int nLen, int m)

{

if(m > nLen)

return;

int\* out = new int[m];

Combination(arr,nLen,m,out,m);

delete [] out;

}

## 方法二：二进制组合算法：

**在ARAIM中使用了该方法进行选星组合，并做了适当的修改：输出二进制组合表**  
 思路是开一个数组，其下标表示1到m个数，数组元素的值为1表示其下标  
 代表的数被选中，为0则没选中。   
 首先初始化，将数组前n个元素置1，表示第一个组合为前n个数。   
 然后从左到右扫描数组元素值的“10”组合，找到第一个“10”组合后将其变为“01”组合，同时将其左边的所有“1”全部移动到数组的最左端（只有第一位变为0才需要移动，否则其左边的1本来就在最左端，无需移动）。   
 当第一个“1”移动到数组的m-n的位置，即n个“1”全部移动到最右端时，就得到了最后一个组合。   
 例如求5中选3的组合：   
 1 1 1 0 0 //1,2,3   
 1 1 0 1 0 //1,2,4   
 1 0 1 1 0 //1,3,4   
 0 1 1 1 0 //2,3,4   
 1 1 0 0 1 //1,2,5   
 1 0 1 0 1 //1,3,5   
 0 1 1 0 1 //2,3,5   
 1 0 0 1 1 //1,4,5   
 0 1 0 1 1 //2,4,5   
 0 0 1 1 1 //3,4,5

代码实现：

void Combine(int arr[], int n, int m)

{

if(m > n)

return;

int\* pTable = new int[n]; //定义标记buf并将其前m个置1

memset(pTable,0,sizeof(int)\*n);

for(int i = 0; i < m; ++i)

pTable[i] = 1;

bool bFind = false;

int tt=0;

do

{

//------输出当前组合的二进制表---------

for (int k = 0; k < n; k++)

{

//cout << pTable[k] << " ";

AllComb(tt, k) = pTable[k];

}

//cout << endl;

tt++;

//-------------------------------------

for (int i = 0; i < n; i++) //打印当前组合

{

**if(pTable[i])**

**cout << arr[i] << "\t";**

**}**

**cout << endl;**

bFind = false;

for(int i = 0; i < n-1; i++)

{

if(pTable[i]==1 && pTable[i+1]==0)

{

swap(pTable[i],pTable[i+1]); //调换10为01

bFind = true;

if(pTable[0] == 0) //如果第一位为0，则将第i位置之前的1移到最左边，如为1则第i位置之前的1就在最左边，无需移动

{

for (int k=0, j=0; k < i; k++) //O(n)复杂度使1在前0在后

{

if(pTable[k])

{

swap(pTable[k],pTable[j]);

j++;

}

}

}

break;

}

}

} while (bFind);

delete [] pTable;

}

# 25 Try throw catch抛出异常的使用

**try throw catch异常抛出处理，类似switch catch,使用try throw catch的好处在于，可以避免程序由于异常而导致的错误，在发生错误前，提前告知用户并终端程序。**

template<class Ta,class Tb,class Tc>

Ta abc(const Ta &a,const Tb &b,const Tc &c)

{

if(a<=0 || b<=0 || c<=0)

throw "all parameters should be >0";

if(a-10<=0.00001)

throw 10;

if(a-20<=0.00001)

throw 1.5;

return a+b\*c;

}

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

{

QCoreApplication mya(argc, argv);

//------------------------------

int a=20,b=1,c=4;

try{cout<<abc(a,b,c)<<endl;}

catch(char \*e)

{

cout<<"aaaa"<<endl;

cout<<"an exception has been thrown"<<endl;

cout<<e<<endl; //抛出的字符串

}

catch(int& i)

{

cout<<"int error "<<i<<endl; //抛出的值

}

catch(double& d) //抛出的值

{

cout<<"double error "<<d<<endl;

}

cout<<"-------------\n";

cout<<"end of program"<<endl;

return mya.exec();

}

# 26 递归函数的用法

template<class T>

T arrsum(T\* a,int n)

{

if(n>0)

return arrsum(a,n-1)+a[n-1];

return 0;

}

template<class T>

T csum(T n)

{

if(n>0)

return n+csum(n-1);

return 0;

}

# 27 STL容器

STL就是Standard Template Library，标准模板库：容器、适配器、迭代器、函数对象和算法的集合。这可能是一个历史上最令人兴奋的工具的最无聊的术语。从根本上说，STL是一些“容器”的集合，这些“容器”有list,vector,set,map等。

1. STL算法accumulate

对顺序表元素按顺序累积求和或其他运算。a[5]={1,2,3,4,5};n=5;

accumulate(a,a+n,thesum); //首元素地址，末元素地址，返回累加值

accumulate(a,a+n,themul,multiplies<T>()); //返回累乘值

1. STL算法copy和next\_permutation

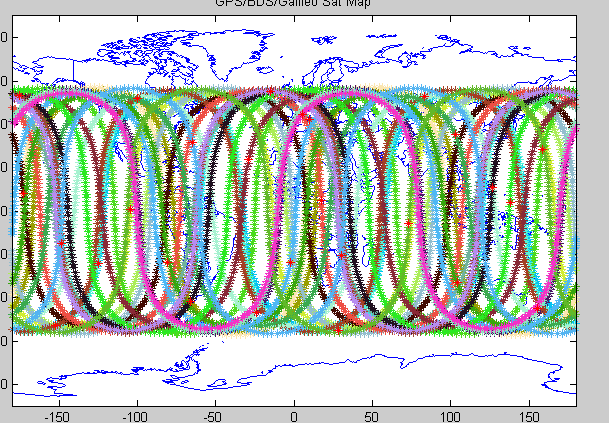
copy(start,end,to);

next\_permutation(start,end);

# 28 matlab笔记

## 1.画随机颜色的图

plot(xC,yC,'\*','color',[rand(),rand(),rand()]);



## 2.矢量和矩阵扩充

矢量扩充：a(end+1,:)=b;

矩阵扩充：A=[A A1];