C语言

# 2.C Primer Plus

阅读《C primer plus中文版第4版》时做的笔记。

## 2.1.文件输入/输出(第13章)

### 2.1.1.fopen()函数

fopen()函数打开文件，在“stdio.h”中声明，包含两个参数：第一个是要打开的文件的路径，字符串形式；第二个是文件打开的模式，同样是字符串模式。关于模式，参见下表：

|  |  |
| --- | --- |
| 模式字符串 | 含义 |
| “r” | 可以读取打开的文件 |
| “w” | 可以向打开的文件写入内容，写入前先将文件清空。文件不存在时，创建 |
| “a” | 向打开的文件写入内容，直接在文件末尾追加，不清空文件。文件不存在时，同样创建文件。 |
| “r+” | 可以读、写打开的文件。*该文件必须存在，否则打开失败。* |
| “w+” | 可以读写打开的文件，如果文件存在就清空文件并添加内容；否则创建文件。 |
| “a+” | 可以读写打开的文件。如果文件存在直接在末尾追加内容，否则先创建文件。**可以读取整个文件，但写入时只能追加。** |
| “rb”,”wb”,”ab”,  ”rb+”,”r+b”,”wb+”,  “w+b”,”a+b”,”ab+” | 类似以上的模式，只是使用二进制模式打开文件，而不是文本模式 |

**需要注意的是**，一旦使用了“w”相关的模式打开文件，文件内容将被清空！应慎重使用！

打开文件成功后，fopen()函数将返回一个文件指针(FILE \*fp, file pointer)，其他文件操作函数通过这个文件指针操作该文件。FILE在“stdio.h”中声明。

打开文件失败，fopen()函数将返回空指针(NULL，stdio.h中定义)。磁盘已满、文件名非法、存取权限不够或者硬件问题都会导致该函数执行失败。

**注意：**FILE \*fp = NULL;

fp = fopen(“test.txt”, “r+”);

fclose(fp);

fp = NULL;

这是应该遵守的编码风格，最后fp指向NULL可以避免野指针的出现，减少内存空间的浪费。

**二进制与文本模式的区别：**

* 在windows系统中，文本模式下文件以“\r\n”代表换行。如果以文本模式打开文件，并执行fputs等函数写入“\n”时，实际写入的是“\r\n”；
* 在linux系统中文本模式下，文件以“\n”代表换行。所以linux系统中文本模式与二进制模式没有任何差别。

**注意：**

**Windows上的路径和Linux上的是不同的表示方法，例如：**

**fopen(“/home/wujl/test.txt”, “r”);**

**fopen(“C:\\folder\\test.txt”, “r”); //这里使用了两个反斜杠，不是直接从目录路径复制过来就能直接使用的**

### 2.1.2.getc()和putc()函数

ch = getc(fp);将fp指向的文件中的一个字符读出，并赋值到ch中。通过：

while((ch = getc(fp)) != EOF) {}

语句，可以以字符为单位遍历整个文件。

putc(ch, fp);语句将字符ch的内容写入到fp指向的文件中。当fp为stdout时，该字符将被输出到标准输出(普遍意义上来说，就是显示器)上，这时候该函数的功能与putchar()函数完全相同。

### 2.1.3.fclose()函数

fclose(fp)关闭文件指针fp指向的文件。如果关闭成功返回0，否则返回EOF。

### 2.1.4.fprintf()和fscanf()函数

#include <stdio.h>

int fprintf(FILE \*stream, const char \*format[, arguments, ...]);

int fsancf(FILE \*stream, char \*format[, arguments, ...]);

fprintf函数将指定格式的内容输出到指定的文件中；fscanf函数从一个流中执行格式化输入，遇到空格和换行时结束输入。**注意，空格也是要结束的，相比较而言，fgets函数遇到空格是不结束的。**

例如：fprintf(fp, “%s%c\n”, str, ch);

fscanf(fp, “%d%s\n”, i, str);

fscanf函数的返回值是整形，表示成功读入的参数个数。

### 2.1.5.fgets()函数

char \*fgets(char \*buf, int bufsize, FILE \*stream);

该函数从文件指针stream中读取数据，每次读取一行。读取的数据保存在buf指向的字符数组中，每次最多读取bufsize-1个字符(第bufsize个字符将自动赋“\0”)。如果文件中的该行不足bufsize个字符，那么读完该行就结束。

函数成功返回buf，失败或读到文件结尾返回NULL。因此直接通过返回值判断函数是否出错时不正确的，应该借助feof函数或者ferror函数来判断。

如果使用fgets()函数读取文件，第一次读取的bufsize是5，而文件的第一行有10个字符(算上”\n”)，那么读取文件的指针会偏移到当前读取完的这个字符之后的位置。也就是说，第二次读取文件的时候，会继续读取其后的字符。

如果使用该函数读取文件的时候bufsize大于该行的字符总数加2(多出来的2个，一个是”\n”，另一个是字符串本身的结束符”\0”)，文件不会继续读取，在读取该行结束后直接结束读取，将文件指针指向下一行的开始位置。

与gets()函数相比，fgets()函数由于制定了bufsize的值，可以一定程度上减少内存越界、堆栈溢出等问题，所以更加安全。

### 2.1.6.fputs()函数

int fputs(char \*str, FILE \*fp);

该函数向指定文件写入一个字符串(不自动写入字符串结束标记符“\0”)。成功写入后，文件指针的位置会随之自动后移，函数返回一个非负整数；否则返回EOF(符号常量，值为-1)。

### 2.1.7.fseek()函数

int fseek(FILE \*stream, long offset, int fromwhere);

函数设置文件指针stream的位置。如果执行成功，stream将指向以fromwhere为基准，偏移offset(可以为负值)个字节的位置；如果执行失败(例如offset超过文件自身大小)，不改变stream的位置。

fromwhere的取值：

* SEEK\_SET，文件头；
* SEEK\_CUR，当前位置
* SEEK\_END，文件尾

执行成功，返回0；否则返回-1，并设置errno的值，可以使用perror()函数输出错误。

### 2.1.8.ftell()函数

long ftell(FILE \*fp);

返回文件指针当前位置相对于文件头的偏移字节数。

利用该函数可以得到一个文件的长度：fseek(fp, 0L, SEEK\_END); len = ftell(fp) + 1;

### 2.1.9.fread()和fwrite()函数

size\_t fread(void \*buffer, size\_t size, size\_t count, FILE \*stream);

该函数从文件中读取数据，最多读取count个元素，每个元素size字节。如果读取成功返回实际读取到的字节数；否则返回0.

size\_t fwrite(const void \*buffer, size\_t size, size\_t count, FILE \*stream);

该函数向文件写入一个数据块buffer，写入成功将返回实际写入的数据项个数count。

C++ Primer

本部分为阅读《C++ primer》的读书笔记。

# 1.变量和基本类型（第2章）

## 1.1.基本类型

Float尽可以表示到小数点后6位的精确度，double一般来讲都最少可以表示到10位以后。

三种数字表示方法：20(十进制)，024(八进制)，0X14(十六进制)。 注意，以“0”开头的表示是八进制的表示方法。

单词true和false是**布尔型**的字面值：bool test = true;

变量初始化C++支持两种形式：复制初始化和直接初始化。复制初始化使用等号(=)，直接初始化把初始化式放在括号里，例如：int a(10)是直接初始化，int a = 10是复制初始化。

在C++中需要理解“**初始化不是赋值**”。初始化是创建变量并给它赋初值，而赋值则是擦除对象的当前值并用新值代替。直接初始化语法更加灵活，而且效率更高。但对于内置数据类型的初始化来说，直接初始化和复制初始化的区别并不大。

## 1.2.变量

**声明和定义**

变量的定义用于为变量分配存储空间，还可以指定初始值。在一个程序中，变量有且只能有一个定义！

声明用于向程序表明变量的类型和名字，可以通过extern关键字声明变量名而不定义它。Extern声明不是定义，所以不分配存储空间。实际上，它只是说明变量定义在程序的其他地方。

程序中的变量可以声明多次，但只能定义一次。

只有当声明也是定义时才可以分配存储空间，例如extern double pi = 3.14;这条语句，尽管使用了extern关键字，但还是分配了存储空间，因为它执行了声明的同时完成了定义操作。但是要注意，**只有当extern声明位于函数外部时，才可以含有初始化式**。

Extern double pi = 3.14; //声明+定义，√

Extern double pi; //声明，√

Extern double pi = 3.1415; //重复定义，×

**Const限定符**

可以通过使用const限定符使一个变量被限制成常量，从而达到不能被修改的目的。注意，**由于修饰后的是一个常量，所以在声明的时候必须初始化**：const int a = 0;是正确的，而const int a;就是不正确的使用方式。

在全局作用域里定义非const变量时，它在整个程序里都可以访问，例如：

//file1.cpp

int a;

//file2.cpp

Extern int a;

…

与这种变量不同，除非特别说明，在全局作用域中声明的const变量时定义该对象的文件的局部变量。此变量只存在于那个文件中，不能被其他文件访问。

所谓特殊说明即在声明时就是用extern关键字说明可以外部访问，例如：

//file1.cpp

Extern Const int a = 0;

//file2.cpp

Extern const int a;

…

**枚举**

枚举的基本使用如下：enum week {Monday, tusday, sunday};默认的，第一个枚举成员赋值为0，后面的每个枚举成员比前面的大1.

枚举成员是常量。枚举成员的值可以相同，例如：enum points {a = 2, b, c= 3, d};这样定义下b和c的值显然都是3，是正确的定义方式。

每个enum都定义了一个新的类型。和其他类型一样，可以定义和初始化points类型的对象，也可以以不同的方式使用这些对象。枚举类型的对象的初始化和赋值，只能通过枚举成员或同一枚举类型的其他对象来进行：

Points p1 = a; //这是正确的赋值方式

Points p2 = 3; //不正确

P2 = Monday; //不正确，Monday不属于points这个枚举类型

P2 = b; //这样赋值是可以的，以points中的一个成员赋值

**头文件**

头文件一般包含：类的定义，extern变量的声明和函数的声明。

# 2.标准库类型(第3章)

## 2.1.标准库string类型

string类型支持长度可变的字符串，使用前应：

#include <string>

using std::string;

### 2.1.1.string对象的定义和初始化

string s1; //s1声明为一个空的字符串

string s2(s1); //s2声明为s1的副本；

string s3(“value”); //s3是值为value的字符串；

string s4(n, ‘c’); //s4初始化为字符c的n个副本

### 2.1.2.string对象的操作

#### size和empty操作

size操作获得string对象的长度，也就是该对象包含的字符的个数。该操作返回的是string::size\_type类型的结果，该类型是为了兼容各种机器的硬件而出现的。虽然可以直接使用int等类型存储size操作的结果，但仍旧建议使用该类型。例如：

for(string::size\_type ix = 0; ix < str.size(); ix++) {}

可使用empty操作判断string对象是否为空，返回true说明为空，否则不空。

#### 关系操作符

可直接使用“==”、“！=”、“<=”、“>=”、“<”、“>”等操作符直接对两个string对象进行比较操作，比较的规则与C语言strcmp()函数相同。

#### 赋值

string s1, s2;

s2 = “test”;

s1 = s2;

最后一句将s2赋值给s1.

#### 两个string对象相加

支持“+”和“+=”运算符，将两个string对象的内容进行连接，例如：

string s1(“Hello”), s2(“ world\n”);

string s3 = s1 + s2;

s1 += s2;

#### string对象和字符串字面值的连接

可以如下连接：

string s1(“Hello”), s2(“world”);

string s3 = s1 + “ ” + s2 + “\n”;

**注意：执行此类操作时，“+”操作符的左右操作数必须至少有一个是string对象！**

例如，该表达式就是不对的：string s4 = “Hello” + “ world\n”；因为它直接将两个字面值进行了连接，操作符的左右没有至少一个string对象。

##### 从string对象获取字符

string类型支持通过下标操作符[]访问其中的每个字符，下标需要使用size\_type类型的值来确定需要访问的字符的位置，例如：

string::size\_type idx = 1;

string s = “string test”;

cout << s[idx] <<endl;

当index超出下标范围时将引发溢出错误，所以一般应配合string.size()操作。

下标操作可以做左值，例如：s[idx] = ‘\*’;操作时允许的，将改变指定index的字符。

### 2.1.3.string对象中字符的处理

这些字符处理函数都在cctype头文件中定义。

|  |  |
| --- | --- |
| 函数名 | 功能 |
| isalpha(c) | 指定字符是否是字母 |
| isalnum(c) | 指定字符是否是数字或字母 |
| isdigit(c) | 指定字符是否是数字 |
| isxdigit(c) | 指定字符是否是十六进制数字 |
| tolower(c) | 指定字符转换成小写，如果本身就是小写不做改变 |
| toupper(c) | 指定字符转换成大写，如果本身就是大写不做改变 |
| islower(c) | 指定字符是否是小写字符 |

上面的判断类型的函数，返回0表示不满足判断条件，返回一个非零值(无任何意义)表示满足判断条件。

## 2.2.标准库vector类型

vector是同一种类型的对象的集合，每个对象都有一个对应的整数索引值。使用vector前需要包含头文件：

#include <vector>

using std::vector;

vector不是一种数据类型，而只是一种类模板，可以用来定义任意多种数据类型。例如，vector<int>和vector<string>就都是数据类型。

### 2.2.1.vector对象的定义和初始化

vector<T> v1; //默认构造函数，v1为空

vector<T> v2(v1); //v2是v1的副本

vector<T> v3(n, i); //v3初始化为包含n个元素i的vector对象

vector<T> v4(n); //n个该类型默认初始值的vector对象， 例如int的默认初始是0

注意：vector对象的动态增长效率极高，所以一般的使用方式是先初始化一个空的vector对象，再动态的向其中增加元素。

### 2.2.2.vector对象的操作

##### size和empty

类似于string对象，vector对象同样有size和empty函数，前者统计vector中有多少元素，后者判断是否为空。

size函数返回值是vector<T>::size\_type，使用同string。

注意，使用size\_type类型时，必须包含元素类型，类似于vector::size\_type的表述是错误的，必须是类似于vector<int>::size\_type的表述才是正确的。

##### 向vector添加元素

push\_back()函数用来接收一个新的元素，并将它插入到vector对象的后面，例如：

string word;

vector<string> text;

while(ci >> word) {

text.push\_back(word);

}

##### 下标操作

vector同样支持下标操作，类似于string类型的使用，可以读取指定下标的元素值。

但要向vector中增加元素的时候，必须使用push\_back函数，要直接对空的vector通过下标引用的方式进行赋值，是无法成功的。

必须是已存在的元素才能用下标操作符进行索引，通过下标操作进行赋值时不会添加任何元素。

## 2.3.迭代器iterator

除了使用下标可以访问到vector中的元素外，标准库还提供了另一种方法：迭代器。迭代器是一种检查容器内元素，并遍历元素的数据类型。

注意：标准库为每一种标准容器定义了对应的迭代器类型。并不是所有的容器都支持下标操作，但所有的容器都支持迭代器。现代C++更倾向于使用迭代器而不是下标访问容器元素，即使对支持下标操作的容器也是如此。

### 2.3.1.容器的iterator类型

每种容器都有自己的迭代器类型，例如vector相关的定义可以是：

vector<int>::iterator iter\_vect\_int；

这行语句就定义了一个名为iter\_vect\_int的迭代器，数据类型是由vector<int>定义的iterator类型。

### 2.3.2.begin和end操作

如果容器中有元素，begin()返回的迭代器指向第一个元素，例如：vector<int>::iterator iter = ivec.begin()；假设vector类型的ivec不空，iter就是ivec[0]。

由end操作返回指向容器的“末端元素的下一个”。

如果容器为空，begin和end指向的将是同一个对象。

end操作返回的迭代器并不指向vector中任何实际的元素，而只是起到哨兵(sentinel)的作用，标明我们已经处理完vector中的所有元素。

### 2.3.3.自增和解引用运算

迭代器可以使用解引用操作符(\*)访问迭代器指向的元素。假设iter指向vector对象ivec的第一个元素，那么\*iter和ivec[0]就指向同一个元素。语句“\*iter = 0”就是将这个元素的值赋0.

迭代器使用自增元素符向前移动，指向下一个元素，类似于指针的概念。例如：

for(vector<int>::iterator iter = ivec.begin(); iter != ivec.end(); ++iter) {

\*iter = 0;

}

该操作可以使用下标操作完成，但是现代C++建议使用上面的迭代器的方式实现。

### 2.3.4.迭代器的算术操作

除了自增操作外，vector迭代器(**其他容器迭代器很少支持该操作！**)还支持其他的算术运算，最常用的是：iter + / - n；该操作将直接将iter指向当前位置之前/之后n个元素的元素。

例如，vector<int>::iterator mid = ivec.begin() + ivec.size() / 2;将直接使迭代器指向最接近vector中间元素的位置。

## 2.4.标准库bitset类型

Bitset类用来处理二进制格式的数据，使用该类型需包含：

#include <bitset>

Using std::bitset;

### 2.4.1.初始化bitset对象

bitset<n> b; //b有n位，每一位都默认初始化为0

bitset<n> b(u); b是unsigned long型u的一个副本

bitset<n> b(s); b是string对象s中含有的位串的副本

bitset<n> b(s, pos, n); b是s中从pos位置开始的n个位的副本

**用unsigned值初始化bitset对象**

可以直接用unsigned long型的值作为bitset对象的初始值，该值将转化为二进制的位模式。

* bitset类型长度大于unsigned long值的二进制位数，多出的高位置0；
* bitset类型长度小于unsigned long值的二进制位数，只使用其低阶位，超过bitset长度的高阶位直接被丢弃；

例如，32位机器上，unsigned long型占用32个bit，那么：

* bitset<16> b1(0xffff); //b1有16个bit，而unsigned long有32个bit，所以截断，只将十六个1赋值到bitset中；
* bitset<32> b2(0xffff); //b2有32个bit，恰好存放unsigned long的值，所以b2[0]…b2[15]都是1，而b[16]…b[31]都是0；
* bitset<128> b3(0xffff); //b3有128个bit，长度大于unsigned long的长度，所以[0…31]存放的是0xffff的值，而其余高位存放的是0；

**使用string对象初始化bitset对象**

当用string对象初始化bitset对象时，string对象直接表示成位模式。从string对象读入位集的顺序是**从右向左**：

String str(“1100”);

Bitset<32> bitvec(str);

Bitvec的位模式第二和第三位都是1，其余位置都是0.如果string对象的字符个数小于bitset类型的长度，高阶位直接置为0.

不一定把整个string对象都作为bitset对象的初始值。相反，可以只用某个子串作为初始值：

String str(“1111111000000011001101”);

Bitset<32> bitvc5(str, 5, 4);

Bitset<32> bitvec6(str, str.size() - 4);

将使用str从str[5]开始的包含4个字符的子串，初始化bitvec5。从str[5]开始的4个字符分别是1、1、0、0，所以bitset从0到3是0011，剩余位都是0。

如果省略了第三个参数，意味着从开始位置一直到string对象的所有字符将初始化bitset。所以bitvec6将使用str的后4位初始化，故此bitvec第0到3位将是1011，其余位都是0.

### 2.4.2.bitset支持的操作

|  |  |  |
| --- | --- | --- |
| 函数 | 含义 | 说明 |
| b.any() | b中是否存在置为1的二进制位 | 有1返回true |
| b.none() | b中是否全是二进制0 | 全0返回0 |
| b.count() | 统计b中二进制1的个数 | 返回值类型为size\_t，该类型定义在cstddef头文件中 |
| b.size() | 计算b中二进制位的个数 | 返回值同样是size\_t类型 |
| b(pos) | 获得b中pos位置的二进制位的值 |  |
| b.test(pos) | 明确b中pos位置的二进制位是否是1 | 返回值是bool类型的true或false |
| b.set() | 设置b中所有位为1 |  |
| b.set(pos) | 设置b中pos位置的二进制位为1 |  |
| b.reset() | 设置b中所有位为0 |  |
| b.reset(pos) | 设置b中pos位置的二进制位为0 |  |
| b.flip() | 对b中所有二进制位取反 |  |
| b.flip(pos) | 对b中pos位置的二进制位取反 |  |
| b.to\_ulong() | 将b转换为unsigned long类型值返回 | 如果bitset对象包含的二进制位数超过u long的长度，运行时异常！ |

## 2.5.数组和指针

### 2.5.1.指针和const限定符

#### 2.5.1.1.指向const对象的指针

如果指针指向const对象，不允许使用指针改变它指向的值。为了保证这一点，C++强制要求指向const对象的指针必须也具有const属性：

const double \*cptr；

这里声明的cptr是一个指向const double类型的指针，const限制的是cptr所指向的对象，而不是cptr。cptr可以指向另外一个const的double对象，但不允许通过cptr修改所指向对象的值，例如：\*cptr = 40.00;该语句试图通过指针修改所指向对象的值的方法，是行不通的。

* 把一个const对象的地址赋给一个普通的、非const对象的指针也会有编译错误：const double pi = 3.14; double \*ptr = &pi的赋值方式不被接受，必须是类似const double \*cptr = &pi才可以；
* 不能使用void\*指针保存const对象的地址，必须使用const void \*类型的指针：const int a = 10; const void \*cp = &a;是正确的赋值方式；
* 允许将非const对象的地址赋给指向const对象的指针：double d = 3.14; const double \*cptr = &d;是可以执行的，但无法通过cptr这个指针修改d的值！

#### 2.5.1.2.const指针

const指针不同于指向const的指针，它是自身不能被修改！例如：int a = 0; int \*const p = &a;那么p就是指向int型变量的const指针，任何企图给const指针赋值的行为都会导致编译时错误。

const指针同样需要在定义的时候完成初始化操作。

const指针不能被修改，但它指向的值如果不是const类型的，可以利用其进行修改：int a = 10; int \*const p = &a; \*p = 11;将是可以正确执行的。

#### 2.5.1.3.指向const对象的const指针

const double pi = 3.14;

const double \*const pi\_ptr = &pi;

这里就既不能修改pi\_ptr所指向的值，也不能修改其自身的指向。

#### 2.5.1.4.指针和typedef

string s;

typedef string \*pstring;

const pstring cstr1 = &s;

pstring const cstr2 = &s;

string \*const cstr3 = &s;

三种定义方式的含义是完全相同的，都是定义一个const指针，指向一个string类型的对象。

### 2.5.2.C风格字符串

#### 2.5.2.1.创建动态数组

C语言使用一对标准库函数malloc和free在自由存储区(堆)上分配存储空间，而C++使用new和delete实现同样的功能。

##### 动态数组的定义

int \*pia = new int[10];

该语句分配了一个含有10个int型元素的数组，并返回指向该数组的第一个元素的指针，这个返回值初始化了指针pia。

##### 初始化动态分配的数组

动态分配的数组，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，否则无初始化：

string \*psa = new string[10];

int \*pia = new int[10];

第一个数组是string类型的，在分配空间后将直接调用string类型的默认构造函数依次初始化数组中的元素；第二个数组是int类型的，内置类型默认没有构造函数，所以数组没有初始化。

也可以在数组长度后加一对空的圆括号，对数组元素做初始化：

int \*pia = new int[10]();

圆括号要求编译器对数组做初始化，在这里就是将数组设置成全0的。但注意，动态分配的数组，采用这种方式，**只能初始化成为这种类型的默认值，不能使用自己定义的值进行初始化。**

##### 动态空间的释放

动态分配的内存在使用完后必须显示的调用释放函数delete执行释放操作，否则内存将逐步耗尽。例如：delete []pia;将释放pia动态分配的空间，注意：中间的一对方括号是必不可少的，否则将只释放pia指向的一个内存单元，其他单元将得不到释放而造成内存泄露。

#### 2.5.2.2.新旧代码的兼容

##### 混合使用标准库string类和C风格字符串

可以把C风格字符串用在任何可以使用字符串字面值的地方：

* 可以使用C风格字符串对string对象进行初始化或赋值；
* string类型的加法需要两个操作数，可以使用C风格字符串作为其中一个，也可以将C风格字符串用作复合赋值操作的右操作数；

反之则不成立：在要求C风格字符串的地方不可以直接使用string类型对象。例如：char \*ch\_arr = str;的操作就是不成立的，要执行此类操作需要调用string类提供的标准库函数：

const char \*ch\_arr = str.c\_str();

注意，c\_str()函数返回的是const char \*的类型。

##### 使用数组初始化vector对象

要使用数组初始化vector对象，必须指出用于初始化的第一个元素以及最后一个元素的下一个位置的地址：

const size\_t arr\_size = 6;

int int\_arr[arr\_size] = {0, 1, 2, 3, 4, 5};

vector<int> ivec(int\_arr, int\_arr + arr\_size);

这样就可以将int\_arr整个数组复制到ivec中。当然也可以只复制一部分数组内容：

vector<int> ivec2(int\_arr + 1, int\_arr + 4);

这个初始化创建了含有三个元素的ivec，分别是int\_arr[1..3]的副本。

# 3.顺序容器(第9章)

一个容器，就是一些特定类型对象的集合。

顺序容器为程序员提供了控制元素存储和访问顺序的能力。这种顺序不依赖于元素的值，而是与元素加入容器时的位置相对应。相比之下， 关联容器则根据关键字的值来存储元素。

所有顺序容器，在以下两个方面上，都有不同程度的性能折中：

* 向容器中添加/删除元素的代价；
* 非顺序访问容器中元素的代价；

标准库支持的顺序容器，概述如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **容器** | **数据结构** | **访问** | **插入/删除** |
| vector | 可变大小的数组 | 支持快速随机访问 | 在尾部之外的位置，执行插入/删除动作，会很慢 |
| deque | 双端队列 | 支持快速随机访问 | 头部/尾部执行插入/删除很快，其他位置慢 |
| list | 双向链表 | 只支持双向顺序访问 | 任何位置执行插入/删除都很快 |
| forward\_list | 单向链表 | 只支持单向顺序访问 | 任何位置插入/删除都很快 |
| array | 固定大小的数组 | 支持快速随机访问 | 不能添加/删除元素 |
| string | 字符串 | 随机访问快 | 尾部插入/删除元素快 |

容器类型上的操作，分为三种类型：

* 所有容器都支持；
* 只有某一类的容器支持的，例如一些操作只针对顺序容器，另一些只针对关联容器，还有一些只针对无序容器；
* 还有一些操作，只适用于一小部分容器；

## 3.1.所有容器都支持的操作

### 3.1.1.迭代器

迭代器指定了一个范围，可以访问到容器中的元素。

普遍的begin() end()迭代器，可以访问容器中所有元素，其是一个左闭右开区间：[begin, end)；

end指向的是最后一个元素之后的位置；

Begin和end有其他多个版本：以r开头的版本返回反向迭代器；以c开头的版本返回const迭代器；

### 3.1.2.Array具有固定大小

与内置数组一样， array容器的大小也是类型的一部分。定义一个array时，除了类型，还要指明array 的大小，例如：array<int, 10>, array<string, 20>，等；

与其他容器不同，array在创建的时候，就已经将空间分配好了。

### 3.1.3.容器大小计算

成员函数size返回容器中元素的数目，需要注意，**forward\_list不支持该属性**；

Empty函数，返回容器是否为空，如果为空返回true，否则返回false；

Max\_size函数返回一个大于或等于该类型容器所能容纳的最大元素值容器；

### 3.1.4.关系运算符

每个容器类型都支持相等运算符(==, !=)；除了无序关联容器外的所有容器都支持关系运算符（<, >, <=, >=）；

运算符两边的运算对象，必须是相同类型的容器，且必须保证相同类型的元素。例如，vector<int>和vector<string>的比较，vector<int>和list<int>的比较，都是不符合规则的。

容器的比较，规则很简单：

* 大小相等，并且所有元素两两相等，两个容器相等；
* 一个容器是另一个容器的前缀子序列，那么size较小的容器小于size较大的容器；
* 如果不存在前缀子序列的关系，那么比较结果，完全取决于第一个不相同的元素的比较结果，与容器中的元素个数没有直接关联；

需要注意的是，只有容器中的类对象，定义了比较运算符，才可以进行比较！

## 3.2.顺序容器操作

### 3.2.1.插入元素

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **操作名称** | **Vector** | **Deque** | **List** | **Forward\_list** | **array** | **string** |
| c.push\_back(t) | 支持 | 支持 | 支持 | 不支持 | 不支持 | 支持 |
| c.push\_front(t) | 不支持 | 支持 | 支持 | 支持 | 不支持 | 不支持 |
| c.emplace\_back(args) | 支持 | 支持 | 支持 | 不支持 | 不支持 | 支持 |
| c.emplace\_front(args) | 不支持 | 支持 | 支持 | 支持 | 不支持 | 不支持 |
| c.insert(p, t) | 支持 | 支持 | 支持 | 不支持(自有) | 不支持 | 支持 |
| c.insert(p, n, t) | 支持 | 支持 | 支持 | 不支持(自有) | 不支持 | 支持 |
| c.insert(p, b, e) | 支持 | 支持 | 支持 | 不支持(自有) | 不支持 | 支持 |
| c.insert(p, il) | 支持 | 支持 | 支持 | 不支持(自有) | 不支持 | 支持 |
| c.emplace(p, args) | 支持 | 支持 | 支持 | 不支持(自有) | 不支持 | 支持 |

#### Push函数

Push\_back和push\_front函数提供了向顺序容器头部和尾部新增元素的办法。

#### Insert函数

Insert函数可以将元素插入到任意指定位置，接受一个迭代器作为第一个参数，迭代器指定了要在容器中什么位置放置新元素。这个迭代器可以指向容器中任何位置，包括尾后迭代器。也正因为此，新的元素将被插入到迭代器所指定的位置之前。

Insert支持有多个重载函数，说明如下：

* C.insert(p,t)，将t插入到p指定的迭代器位置之前；
* C.insert(p, n, t)，将n个t元素插入到p指定的迭代器位置之前；
* C.insert(p, b, e)，将b和e这对迭代器包含的元素插入到p指定的迭代器之前；注意，**不能将自己的迭代器范围内的元素插入到本容器中**！
* C.insert(p, args)，使用初始化列表插入元素；

Insert的返回值：指向第一个新加入元素的迭代器；

#### Emplace函数

新标准提供了三个emplace函数给顺序容器，分别是:emplace\_back, emplace\_front, emplace；这几个元素并不是拷贝元素，而是构造元素后插入到指定位置；

由于emplace函数在容器中直接构造元素，因此传入的参数必须与元素的构造函数相匹配。

### 3.2.2.访问元素

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **操作名称** | **Vector** | **Deque** | **List** | **Forward\_list** | **array** | **string** |
| c.back() | 支持 | 支持 | 支持 | 不支持 | 支持 | 支持 |
| c.front() | 支持 | 支持 | 支持 | 支持 | 支持 | 支持 |
| c[n] | 支持 | 支持 | 不支持 | 不支持 | 支持 | 支持 |
| c.at(n) | 支持 | 支持 | 不支持 | 不支持 | 支持 | 支持 |

front操作返回首元素的引用；back操作返回尾元素的引用；

需要将back与end迭代器做出区别，back返回的是尾元素，是最后一个元素；end迭代器指向的是尾后，是最后一个元素之后的位置；

此外，使用front和back函数时，有必要先通过empty判断容器是否为空。对一个空容器使用这两个函数调用，就像使用一个越界的下标一样，会造成严重的运行时错误。

这几个函数返回的都是引用，如果容器是普通容器，返回的就是普通引用；如果容器是const类型的，返回的就是const类型的应用；例如：auto & v = c.back(); 这时候可以通过改变v的值，直接修改c中尾元素的值；

### 3.2.3.删除元素

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **操作名称** | **Vector** | **Deque** | **List** | **Forward\_list** | **array** | **string** |
| c.pop\_back() | 支持 | 支持 | 支持 | 不支持 | 不支持 | 支持 |
| c.pop\_front() | 不支持 | 支持 | 支持 | 支持 | 不支持 | 不支持 |
| c.erase(p) | 支持 | 支持 | 支持 | 不支持(自有) | 不支持 | 支持 |
| c.erase(b, e) | 支持 | 支持 | 支持 | 不支持(自有) | 不支持 | 支持 |
| c.clear() | 支持 | 支持 | 支持 | 支持 | 不支持 | 支持 |

删除元素会改变容器的size，array是固定size的，因此不支持；

Pop\_back和pop\_front操作返回的都是void，如果需要使用元素的值，就需要首先访问他们(3.2.2.)，在删除他们；

Erase操作可以删除一个迭代器指向的元素，也可以删除一对迭代器包含的元素，删除之后都返回一个迭代器，这个迭代器指向删除的最后一个元素之后的位置；

需要注意，erase删除一对迭代器指定的范围，是[b, e)的，也就是说，删除操作之后，e指向的迭代器将会作为返回值返回。这样就可以理解：c.erase(c.begin(), c.end()) 与 c.clear()等价了，因为erase执行完之后，返回值会指向尾后迭代器，而并不是将尾后删除，指向尾后的尾后。

### 3.2.4.Forward\_list的一些特殊操作

如上所属，forward\_list不支持一些通用的顺序容器的操作，如insert、emplace、erase，其自实现了一些其他函数来支持这些操作。需要自实现的根本原因，在于forward\_list自身单向链表的数据结构。

forward\_list提供了两个访问首前元素的迭代器：lst.before\_begin(), lst.cbefore\_begin();这两个函数返回首前迭代器，指向的是不存在的元素，因此不能解引用；这个迭代器的存在，允许我们在forward\_list的首元素之前插入元素；

forward\_list提供了几个插入元素的函数：lst.insert\_after(p, t), lst.insert\_after(p, n, t), lst.insert\_after(p, b, e), lst.insert\_after(p, il), lst.emplace\_after(p, args)；这几个函数都是在p后插入若干个新元素，返回指向最后一个插入元素的迭代器；如果p本身是尾后迭代器，那么行为就不可预估了；

forward\_list提供了几个删除元素的函数：lst.erase\_after(p), lst.erase\_after(b, e)；删除p之后的元素，或者删除b之后(不包含b)直到e之间的元素。返回一个指向被删除元素之后元素的迭代器，如果不存在这样的元素，就返回尾后迭代器。如果p指向了lst的尾元素或者是一个尾后迭代器，那么函数行为未定义。

### 3.2.5.改变容器大小

改变容器大小，显然array不可以支持。

改变容器大小的规则是：如果是增加容器size，有新元素添加到容器末尾；如果是减少容器size，删除末尾的元素。

支持两种格式：c.resize(n), c.resize(n, t)；如果是新增元素，前者会使用默认值赋值新增加的元素，后者会使用t赋值；

### 3.2.6.容器操作可能使迭代器失效

由于向容器中添加或删除元素，可能导致迭代器失效，因此必须保证每次改变容器的操作之后，都重新定位迭代器。这个建议对于vector、string和deque尤为重要。

很重要的一个原则是：不要保存end返回的迭代器。例如：

Auto end = vec.end();

While(begin != end) //更应该使用:while(begin != vec.end()),每次都重新获取一次尾后迭代器;

{

...;

}

如果函数体中，做出了insert、erase等操作，导致了容器的迭代器失效，那么vec.end()是会发生改变的，使用变量end获取到的值将不再是容器的尾后迭代器的位置，那么这个程序就不再是正确的逻辑语义了。

### 3.2.7.Vector对象是如何增长的

Vector和string是典型的顺序存储、空间可扩展的容器，为了满足容器访问的高效性，不能每次添加元素时都重新分配一次内存，普遍的做法是一次分配比新的空间需求更大的内存空间，这些多出来的空间作为备用，以保证操作元素的速度。

TODO， size(), capacity(), shrink\_to\_fit(), reserve()

## 3.3.额外的string操作

除了顺序容器共同的操作之外，string类型还提供了一些额外的操作。这些操作中的大部分，要么是提供string类和C风格字符数组之间的转换，要么是增加了允许我们使用下标代替迭代器的版本。

### 3.3.1.其他几个构造函数

String还支持另外几种构造函数的定义方式：

* String s(cp, n):cp是一个数组，s是cp指向的数组中的前n个字符的拷贝；该数组应该至少包含n个字符！
* String s(s2, pos2):s是string s2从pos2开始的字符的拷贝；如果pos2>s2.size()，那么构造函数的行为未定义；
* String s(s2, pos2, len2):s是string s2从pos2开始的len2个字符的拷贝。如果pos2>s2.size()，那么构造函数的行为未定义；不管len2的值多大，构造函数最多只拷贝s2.size() - pos2个字符；

需要格外注意的是第一种构造函数格式，cp是一个字符数组，如果使用string s(cp, n)的格式做构造函数，不要求cp一定以’\0’结尾，s会自己添加；但如果使用的是string s(cp)的方式定的，那么就一定要以’\0’结尾，否则函数行为未定义。例如：

Char a[] = {‘h’, ‘e’};

String s1(a, 2); //正确，拷贝2个字符到s1，s1会自己添加结束符

String s2(a); //错误，a不是以’\0’结尾，s2的构造函数结果未定义；

### 3.3.2.Substr操作

Substr操作返回一个string，他是原string的一部分或者全部的拷贝。

函数原型：S.substr(pos, n)

返回值：返回一个string，包含了s中从pos开始的n个字符的拷贝；

参数说明：

* Pos，默认值是0，设置后表示从那个字符开始拷贝；
* N，默认值是s.size() - pos，表示拷贝拷贝的长度；

如果pos的值超出了string的长度，那么substr会抛出out\_of\_range的异常；如果要拷贝的字符串超出了string的长度，那么substr会调整计数值n，只拷贝到string末尾。

### 3.3.3.改变string的其他方法

String额外支持了一些函数，概述两个觉的有用的：append和replace；

S.append(args)：将args追加到s，返回一个s的引用；

S.replace(pos, len, args)：删除s中从pos开始的len个字节的内容，替换为args的内容，args的长度可以和len不相同；

### 3.3.4.String搜索操作

String类提供了6个不同的搜索函数，每个函数有4个重载版本，见下表：

|  |  |  |
| --- | --- | --- |
| 6个搜索函数 | | |
| S.find(args) | | 搜索s中args第一次出现的位置 |
| S.rfind(args) | | 搜索s中args最后一次出现的位置 |
| S.find\_first\_of(args) | | 搜索s中args任意一个字符第一次出现的位置 |
| S.find\_last\_of(args) | | 搜索s中任意一个字符最后一次出现的位置 |
| S.find\_first\_not\_of(args) | | 搜索s中第一个不在args中的字符 |
| S.find\_last\_not\_of(args) | | 搜索s中最后一个不在args中的字符 |
| 4个重载版本都针对args的格式 | | |
| c, pos | 从s中位置pos开始查找字符c，pos默认为0 | |
| s2, pos | 从s中位置pos开始查找字符串s2，pos默认为0 | |
| cp, pos | 从s中pos开始查找cp指向的C风格字符串，pos默认为0 | |
| cp, pos, n | 从s中位置pos开始，查找cp指向的数组的前n个字符，pos和n没有默认值 | |

搜索函数返回一个string::size\_type值，表示匹配发生的下标。**如果搜索失败，返回一个string::npos的static成员**。

String::size\_type是一个unsigned类型，因此，用一个int或其他有符号类型来保存这些函数的返回值不是一个好主意。最直观的是，string::npos在标准库中定义为-1，但由于string::size\_type是unsigned，因此其实npos的取值是string最大的可能大小。

### 3.3.5.Compare函数

除了关系运算符，string类还提供了一组compare函数，这些函数与C标准库中的strcmp函数很类似。

其返回值，根据s是等于、大于还是小于参数指定的字符串，s.compare返回0、正值或者负数。

String重载的比较运算符(==, !=, >, <, 等等)，string的compare函数，c语言的strcmp函数，比较规则都相同：

* 内容和长度都完全相同，相等；
* S1是s2的子串的话，s1小于s2；
* 从第一个字符开始比较，直到第一个不相同的字符出现，谁的这个字符的ascii码值更大，谁大；

注意，在不相等、且不存在子串的情况下，并不比较长度，而是依赖于第一个不相等的字符的大小来判断大小；

|  |  |
| --- | --- |
| s.compare的几种参数形式 | |
| S.compare(s2) | 比较s和s2 |
| S.compare(pos1, n1, s2) | 将s中从pos1开始的n1个字符与s2比较 |
| S.compare(pos1, n1, s2, pos2, n2) | 将s中从pos1开始的n1个字符，与s2中从pos2开始的n2个字符比较 |
| S.compare(cp) | 将s与cp指向的C语言风格字符串比较 |
| S.compare(pos1, n1, cp) | 将s中pos1开始的n1个字符，与cp指向的字符串比较 |
| S.compare(pos1, n1, cp, n2) | 将s中pos1开始的n1个字符，与cp指向的、开始n2个字符构成的字符串比较 |

需要知道的是，运算符重载实现的比较运算，其实质还是调用了compare函数。如果编译器将模板重载实现为了inline类型，那么运算符重载和compare没有任何性能差异；但如果没有inline实现，那么compare要快一些，因为重载还要过一次函数调用；

### 3.3.6.数值转换

String和数值的转换，类似于atoi等C语言的函数调用，遵循以下规则：要转换为数值的string 的第一个非空白字符必须是数值中可能出现的字符；

如果string不能转换为字符，函数将抛出一个invalid\_argument的异常；如果转换得到的数值无法用任何类型表示，将抛出一个out\_of\_range的异常。

|  |  |
| --- | --- |
| String和数值的转换 | |
| to\_string(value) | 返回数值Value的string表示。这是一组重载函数。Value可以是任何数值类型，int，float等，都有各自的实现； |
| stoi(s, p, b) | 返回s的表示整数内容的起始子串的数值。  p是一个指针，指向一个size\_t类型，用来保存s中第一个非数值字符的下标值。p默认值为0，也就是空指针，表示不保存该值；  b表示转换使用的基数，模式是10，代表十进制； |
| stol(s, p, b) |
| stoul(s, p, b) |
| stoll(s, p, b) |
| stoull(s, p, b) |
| stof(s, p) | 返回s的表示浮点数内容的起始子串的数值；  参数p的作用同上； |
| stod(s, p) |
| stold(s, p) |

## 3.4.容器适配器

除了顺序容器自身，标准库还定义了三种顺序容器的适配器：stack，queue和priority\_queue。

适配器是标准库中的一个通用概念。容器、迭代器和函数都可以有适配器。本质上来说，适配器是一种机制，可以使一种事物的行为看起来像另外一种。所有容器适配器都支持的操作如下表：

|  |  |
| --- | --- |
| 所有容器适配器都支持的操作和类型 | |
| Size\_type | 足以保存当前类型的最大对象的大小 |
| Value\_type | 元素的类型 |
| Container\_type | 实现适配器的底层容器的类型 |
| A a; | 创建一个名为a的空适配器 |
| A a(c); | 创建适配器a，带有容器c的一个拷贝 |
| 关系运算符 | 每个适配器都支持所有的关系运算符：==，!=，>, <, >=, <=等；这些运算符返回底层容器的结果。 |
| 1. empty() | 判断a是否为空 |
| A.size() | 返回a中的元素的数目 |
| Swap(a, b) | 交换a和b的内容，a和b必须有同样的类型，包括底层容器也必须要相同 |
| A.swap(b) | 同上 |

### 3.4.1.定义一个适配器

每个适配器都支持两种构造函数：

* 默认构造函数，创建一个空对象；
* 拷贝另一个容器来初始化适配器，例如deq是一个deque<int>，那么就可以：stack<int> stk(deq);来对stk进行初始化。

默认情况下，stack和queue都是基于deque实现的；priority\_queue是基于vector实现的。我们也可以使用显示指定的方式，更改适配器使用的底层容器类型，例如：stack<int, vector<string>> stk；这种定义方式，就将一个stack指定为基于vector来实现的，而不是默认的deque了。

对于一个适配器，可以使用哪些容器来进行构造是有限制的：

* 不能基于array实现适配器，因为所有适配器都要求能够添加和删除元素；
* 不能使用forward\_list构造适配器，因为所有的适配器都要求具有添加、删除以及访问尾元素的能力；
* Stack是后进先出，只要求具有：push\_back, pop\_back, back操作的支持，因此除去forward\_list和array，其他顺序容器都可以用来构造stack；
* Queue是先进先出，要求具有:push\_back, push\_front, back, front操作，因此可以构造在list和deque之上，但不能构造在vector之上；
* Priority\_queue要求具有front、push\_back、pop\_back的能力，还要求有随机访问元素的能力，因此list不能使用，只能基于vector或deque实现；

各个容器类型支持的适配器类型如下表：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | array | vector | deque | list | Forward\_list |
| Stack | 不支持 | 支持 | 支持(默认) | 支持 | 不支持 |
| Queue | 不支持 | 支持 | 支持(默认) | 支持 | 不支持 |
| Priority\_queue | 不支持 | 支持(默认) | 支持 | 不支持 | 不支持 |

### 3.4.2.栈适配器

如下表格列出了栈支持的几个自有操作：

|  |  |
| --- | --- |
| 栈适配器自有操作 | |
| S.pop() | 删除栈顶元素，但**不返回该元素的值**； |
| S.push(item) | 创建一个新元素并压入栈顶，元素通过拷贝item而来； |
| S.emplace(args) | 创建一个新元素并压入栈顶，元素通过args构造而来； |
| S.top() | 返回栈顶元素，但并不弹出该元素 |

每个容器适配器都定义了自己的一些操作函数，对于这些适配器，只能使用他们定义了的特有操作函数，而不能使用底层容器的一些操作函数。例如，s是基于deque实现的，但是我们只能调用s.push()，而不能使用deque的push\_back()。

我们做一个小示例程序：

#include <stack>

Stack<int> s;

For(size\_t i = 0; i < 10; i++)

S.push(i); //压栈

While(!s.empty())

{

Int value = s.top(); //首元素值的获取，并不出栈

//todo, 使用value

S.pop(); //首元素出栈

}

### 3.4.3.队列适配器

Queue和priority\_queue都定义在头文件<queue>中。下标列出了他们支持的特有操作:

|  |  |
| --- | --- |
| queue适配器自有操作 | |
| q.pop() | 删除队列的第一个元素，但**不返回该元素的值**； |
| q.push(item) | 创建一个新元素并入队列，元素通过拷贝item而来； |
| q.emplace(args) | 创建一个新元素并入队列，元素通过args构造而来； |
| q.front() | 返回首元素，但并不删除该元素 |
| q.back() | 返回尾元素，但并不删除该元素 |
| Priority\_queue适配器自有操作 | |
| q.pop() | 删除队列优先级最高的元素，但**不返回该元素的值**； |
| q.push(item) | 创建一个新元素并放入适当位置，元素通过拷贝item而来； |
| q.emplace(args) | 创建一个新元素并放入适当位置，元素通过args构造而来； |
| q.top() | 返回优先级最高的元素，但并不删除该元素 |

标准库的queue是标准的队列结构，遵循先进先出的存储和访问策略。相比之下，priority\_queue的不同之处在于为元素定义了优先级，新加入的元素，会排在所有优先级比它低的元素之前。

# 4.泛型算法

标准库定义的操作集合很小。标准库并没有给每个容器添加大量的操作，而是提供了一组算法，这些算法中的大多数都独立于特定的容器类型。这些算法的通用性，使得为其取名为泛型算法。

## 4.1.概述

大多数泛型算法都定义在algorithm头文件中，在numeric头文件中，标准库还定义了一些数值泛型算法。

泛型算法本身并不会执行容器的操作，它们只会运行在迭代器之上，执行迭代器的操作。泛型算法的这个特性，保证了：算法永远不会改变底层容器的大小。算法可能改变容器中的元素的值，也可能在容器内移动元素，但不会直接添加或者删除元素。

标准库定义了一种特殊的迭代器：插入器。给这种迭代器赋值的时候，他们会在底层容器上执行插入操作。因此当一个算法操作这样一个迭代器时，迭代器可以完成向容器添加元素的效果。但要注意，这并不是算法完成的，而是插入器完成的。算法本身不会修改容器的大小。

标准库的算法是针对迭代器操作，而不是容器本身，从这个角度理解，也能理解为什么算法不能直接添加/删除元素。

## 4.2.初识泛型算法

### 4.2.1.只读算法

这种算法只会读取其输入范围内的元素，但不改变元素。

#### find算法

查找特定元素的算法，例如：

int val = 42;

auto ret = find(vec.begin(), vec.end(), val);

return ret == vec.end() ? 0 : 1;

find算法返回第一个等于给定值的元素的迭代器，如果没有找到该值，返回第二个参数表示搜索失败。

#### count算法

count函数参数传入与find类似，接受一对迭代器指定的查找范围，和一个值作为被统计的对象。不同的是，其返回的是该值在这个迭代器范围内出现的次数。

#### accumulate算法

该算法用来计算和，例如: int sum = accumulate(vec.cbegin(), vec.cend(), 0);前两个参数指定了要执行加法的迭代器范围，最后一个参数决定了使用哪种加法运算符和返回值的类型。

accumulate将第三个参数作为求和的起点，蕴含了一个编程假设：将元素类型加到和的类型上是可行的。也就是说，迭代器范围指定的序列中的元素类型，必须能和第三个参数匹配，或者能转换为第三个参数的类型使用。

string sum = accumulate(v.cbegin(), v.cend(), string(“”)); //这是正确的，因为string定义了加法操作

string sum = accumulate(v.cbegin(), v.cend(), “”); //这是错误的，因为第三个参数是const char \*的类型，这个类型没有定义加法操作；

#### equal算法

比较两个序列是否保存相同的值的算法。

它依次比较两个序列中的每个元素，只有每个值都相等，才返回true；否则返回false；

该算法接受3个参数：前两个参数表示第一个序列的范围；第三个参数表示第二个序列的首元素，例如：equal(a.begin(), a.end(), b.begin());

equal并不要求两个容器必须是同一类型，只要都能够使用迭代器访问就可以；也不要求元素必须是同一类型，只要能用“==”比较这两个序列的元素就可以。

但是，equal基于一个非常重要的假设：它假定第二个序列至少与第一个序列一样长。

**如果第二个序列5个元素，第一个序列3个元素。前3个元素都相等，结果是什么呢？**

**实测后发现，结果是true。**

### 4.2.2.写容器元素的算法

一些算法将新的值赋予序列中的元素。当我们使用这类算法的时候，要保证序列的大小最少不能小于算法要写入的元素的数目。因为算法本身是不会执行容器操作、修改容器大小的。

#### fill函数

fill函数接受一对迭代器作为输入表示一个范围，接受一个值作为第三个参数。fill函数将这个值赋予迭代器所指向的元素中。例如:

fill(a.begin(), a.end(), 1);

由于fill是向迭代器指定的序列中写入元素，因此只要迭代器有效，那么写入操作就是安全的。

#### fill\_n函数

fill\_n函数接受一个迭代器、一个计数器和一个目标值作为参数传入。将目标值赋予迭代器指向的若干个元素，赋予多少个元素由计数器指定。例如：

fill\_n(a.begin(), 19, 11); //将11赋予a的前19个元素中

这里要注意，fill\_n在给容器赋值的时候，要确保容器的size足够使用。

#### back\_inserter

back\_inserter是一种插入迭代器，可以向容器中添加元素。

通常情况下，我们通过一个迭代器向容器元素赋值时，值被赋予迭代器指向的元素；当我们通过一个插入迭代器赋值时，一个元素将被创建出来，并添加到容器中。

例如：

vector<int> vec;

auto it = back\_inserter(vec);

\*it = 3;

这个过程中，it被创建出来，而且是插入迭代器的类型。赋值给它的同时，会创建一个元素，插入到vec的末尾。

fill\_n(back\_inserter(vec), 10, 0); 这种赋值方式就是可行的，虽然vec最初是空的，但是每次赋值时，back\_inserter都会自动创建元素并添加到容器中。

#### 拷贝算法

copy算法接受三个参数，前两个表示一个输入范围，第三个表示目的序列的开始位置。算法最终将输入范围内的元素都拷贝到目的序列中。这里同样要确保，目的序列的size足以保存输入序列中的元素个数。

例如：auto ret = copy(a.begin(), a.end(), b.begin());

copy函数返回的是目的地址迭代器递增后的值，例如，上例中返回的就是b中尾元素之后的值。

replace函数同样提供了一个拷贝版本。

首先看replace的原始版本：replace(a.begin(), a.end(), 1, 2); 这里会在a中查找所有等于1 的元素，并将其值更换为2；

再看拷贝版本：replace\_copy(a.begin(), a.end(), b.begin(), 1, 2); 这里a就不会发生变化，而是将改变后的值赋予b中。这里要确保b的size足够。当然，也可以通过back\_inserter来保证size：replace\_copy(a.begin(), a.end(), back\_inserter(b), 1, 2);

### 4.2.3.重排容器元素的算法

一些算法会重排容器中元素的位置，一个明显的例子是sort。调用sort会重新排列输入序列中的元素，使之有序，它是利用元素类型的“<”运算符来实现排序的。

例如如下程序，实现将一个vector中的元素排序，并清除重复元素的操作：

vector<string> vec;

genVec(); //为vec填充各个变量，需要自行实现

sort(vec.begin(), vec.end()); //排序，从小到大，有重复的会相邻分布

auto end\_unique = unique(vec.begin(), vec.end()); //将重复的元素移除到vector的尾部，不重复的分布在vector的前半部分，返回值是指向不重复区域之后一个位置的迭代器

vec.erase(end\_unique, vec.end()); //调用容器自己的erase函数，将后半部分重复的数据删除掉。

这里可以看到，sort和unique这些算法并不会删除容器中的元素，只有容器自己的操作才可以真正的删除元素。

## 4.3.定制操作

以sort为例，默认使用的是“<”运算符进行比较并排序，但很多时候我们更希望自己定义比较规则，这时候需要重载sort的默认行为。

### 4.3.1.向算法传递函数

#### 谓词

谓词是一个可以调用的表达式，返回结果是一个能够用作条件的值(boolean)。

标准库支持的谓词有两种：一元谓词和二元谓词。一元谓词意味着它只接受一个参数，二元谓词意味着它有两个传入参数。

接受谓词参数的算法对输入序列中的元素调用谓词。因此，元素类型必须就是、或者能够转换为谓词的参数类型。例如：

bool isShorter(const string & s1, const string & s2)

{return s1.size() < s2.size();}

这里定义了一个函数，传入两个string，返回两者的长度的比较结果。我们可以在排序算法中，使用该谓词，重排容器中元素的顺序。

#### 重载后的排序算法

我们可以使用如上的isShorter作为谓词，重载sort函数，例如：sort(a.begin(), a.end(), isShorter);使用这个方法后，a排序后默认使用size作为规则进行排序。

还可以使用stable\_sort函数，在使用size进行重新排序后，如果size相同的string，还可以按照默认的规则(也就是string的默认比较大小)排序，例如:stable\_sort(a.begin(), a.end(), isShorter);

### 4.3.2.Lambda表达式

由于谓词只接受一个或两个参数，因此希望传递更多参数的时候，谓词无法完成需求。这时候，lambda，是一个很好的选择。

一个lambda是一个可调用的代码单元，可以理解为一个未命名的内联函数。

与其他函数类似，lambda同样具有一个返回类型、一个参数列表和一个函数体。但它的格式与其他普通函数有差别：

[capture list] (parameter list) -> return type {function body}; //[捕获列表] (参数列表) -> 返回类型 {函数体}

capture list是一个lambda所在函数中定义的局部变量的列表，可以被lambda所引用；

parameter list和return type可以不写，但capture list和function body必须存在，否则无法标记这是一个lambda表达式。例如：auto f= [] {return 42;}

不传入明确的返回类型时，编译器自己推断返回类型：如果函数体只有一条语句，该语句是一条return语句，编译器从return的值推断返回类型；否则，编译器都认为返回类型是void。

#### parameter list -- 参数列表

lambda接受传入参数，例如：

[] (const string & a, const string & b) {return a.size() < b.size();}

这个lambda语句和isShorter完成的是同样的功能，传入了两个string类型的参数，比较其size大小。可以在排序算法中直接使用这个lambda表达式：stable\_sort(vec.begin(), vec.end(), [] (const string & a, const string & b) {return a.size() < b.size(); }

#### capture list -- 捕获列表

lambda表达式存在在函数体中，可以使用到函数中的局部变量，需要使用的局部变量需要包含在捕获列表中。例如我们假设在函数中有一个指定的sz值，要寻找所有大于该值的string，那么就有类似的lambda语句：[sz] (const string & a ) {return string.size() < sz; }

lambda捕获参数有两种类型：值捕获和引用捕获。

**值捕获**

值捕获类似于普通的参数传递，其前提是变量可以被拷贝。但与普通的参数传递不同的是，被捕获的变量的值实在lambda创建的时候就传入了，而不是调用的时候传入。例如：

void func()

{

size\_t v1 = 42;

auto f = [v1] {return v1};

v1 = 0; //重新设置变量的值

auto j = f(); //j为42，因为lambda创建的时候其值是42，f保存了我们创建它时v1的拷贝

}

**引用捕获**

与值捕获相对应的，是引用捕获。例如：

void func()

{

size\_t v1 = 42;

auto f = [&v1] {return v1};

v1 = 0; //重新设置变量的值

auto j = f(); //j为0，因为lambda捕获的是v1的引用，v1的改变会同步到这里

}

要注意，使用引用捕获的方式捕获局部变量时，必须保证lambda表达式执行时变量是存在的！

**隐式捕获**

除了显示的指明要引用哪个变量外，还可以隐式的指明，主要是由编译器自己发现我们lambda代码体重使用了哪些变量来决定要捕获哪些变量。为了告诉编译器需要它自行推断捕获列表，要在捕获列表中写一个“&”或者“=”：前者表示引用捕获方式，后者表示值捕获方式。例如：

wc = find\_if(a.begin(), a.end(), [=] (const string & s) {return s.size() < sz;});

这里没有明确指明要引用sz变量，但是捕获列表中使用“=”告知了编译器会有这个需求。编译器自行寻找sz，并加入捕获列表中，以值捕获的方式使用。

如果我们希望对一部分变量采用值捕获，对其他一部分变量使用引用捕获，可以混合使用隐式捕获和显示捕获，例如：

for\_each(words.begin(), words.end(), [&, c](const string &s) {os << s << c;});

for\_each(words.begin(), words.end(), [=, &os] (const string &s) {os << s << c;})

当混合使用隐式捕获和显示捕获时，捕获列表中的第一个元素必须是“&”或“=”。这个符号指定了默认捕获方式是值捕获还是引用捕获。同理，显示捕获的变量必须使用与隐式捕获不同的类型。就是说，如果第一个元素是&，那么显示捕获的变量就必须使用值捕获的方式。

**可变lambda**

使用引用捕获的方式传入的参数，是可以被修改的(const除外)；对于值捕获的方式，默认情况下不能修改其值，但也有手段可以改变其值，这就是增加mutable关键字，例如：

size\_t v1 = 42;

auto f = [v1] () mutable {return ++v1};

v1 = 0;

auto j = f(); //这时候返回值是43，因为v1被改变了

**指定lambda返回类型**

只有一条return语句的lambda函数体，根据return语句的返回值类型，推断其lambda表达式的返回值类型；但是如果函数体中不只是一条return语句，那么编译器默认认为它是返回void类型的。例如：

transform(a.begin(), a.end(), a.begin(), [] (int i) {return i < 0 ? -i : i;}); //只有一条return语句作为lambda表达式的body，正确

transform(a.begin(), a.end(), a.begin(), [] (int i) {if(i < 0) return -i; else return i;}); //错误，有多条语句组成的lambda的函数体，默认返回值类型是void，不能作为赋值用

要解决这类问题，就需要显示指定lambda表达式的返回值，例如：transform(a.begin(), a.end(), a.begin(), [] (int i) **-> int** {if(i < 0) return -i; else return i;}); //这里显示指定了lambda表达式的返回值是int类型的，就可以被正常使用了。

# 5.关联容器

关联容器和顺序容器有着本质上的不同：关联容器中的元素是按照关键字(key)来保存和访问的；而顺序容器中的元素是按照他们在容器中的位置来顺序保存和访问的。

标准库提供8个关联容器，不同之处体现在3个维度上：

* + 每个关联容器或者是一个map，或者是一个set；
  + 或者要求不重复的关键字，或者不要求；
  + 或者按顺序保存元素，或者无序保存元素；

允许重复关键字的容器，名字中都包含multi关键字；不保持关键字按顺序存储的容器的名字中都带有一个unordered关键字。例如：unordered\_multi\_set就是一个无序的、允许重复关键字的set。

|  |  |
| --- | --- |
| map | map |
| set | set |
| multimap | 允许关键字重复的map |
| multiset | 允许关键字重复的set |
| unordered\_map | 无序的map，用哈希函数组织 |
| unordered\_set | 无序的set，用哈希函数组织 |
| unordered\_multimap | 允许关键重复的无序map，用哈希函数组织 |
| unordered\_multiset | 允许关键字重复的无序set，用哈希函数组织 |

## 5.1.关联容器概述

关联容器，不论是有序的还是无序的，都支持普通的容器操作：迭代器，size()，关系运算符等。

但不支持顺序容器的位置相关的操作，例如push\_front等，因为关联容器中的元素并不是按照元素顺序存储，而是按照关键字存储，因此这些操作并无意义。同理，关联容器也并不支持构造函数或插入操作这些接受一个元素值和一个数量值的操作。

关联容器的迭代器都是双向的。

### 5.1.1.定义关联容器

map<string size\_t> a; //定义了一个空容器,关键字类型是string，值类型是size\_t

set<string> b = {“ss”, “cc”}; //定义了一个set，值默认初始化

map<string, string> c = { {“12”, “34”}, {“23”, “45”}}; //定义了一个map，初始化过的

### 5.1.2.关键字类型的要求

关联容器对于关键字类型有限制，无序容器的关键字要求后续详述，有序容器的关键字类型必须定义元素比较的方法。默认情况下，标准库使用关键字类型的“<”运算符来比较两个关键字。

我们可以自定义一个操作，来替代默认的“<”运算符为容器执行排序。所提供的操作，必须在关键字类型上定义一个**严格弱序**，这种操作必须具备以下几个特征：

1. 两个关键字不能同时“小于等于”对方；
2. 关系具备传递性，例如k1小于等于k2，k2小于等于k3，那么k1一定小于等于k3；
3. 如果存在两个关键字，任何一个都不“小于等于”另一个，那么就说这两个关键字是等价的；

要使用如上自定义的比较操作，必须在定义关联容器的时候提供这个操作，例如：

bool compareIsbn(const Sales\_data &a, const Sales\_data &b) {return a.isbn() < b.isbn();}

multiset<Sales\_data**, decltype(comparsIsbn) \***> bookstore(**compareIsbn**);

### 5.1.3.pair类型

pair类型定义在头文件utility中。

pair也是一个模板，一个pair保存两个数据成员，例如：pair<string, string> a;

pair的默认构造函数对数据成员内进行值初始化。如上的定义中，a的两个成员都会被初始化为空字符串。也可以初始化的时候指定值：pair<string, string> b{“c”, “d”};

pair的两个数据成员都是public的，分别是first和second，可以直接访问到这两个成员。

## 5.2.关联容器操作

关联容器定义了几个额外的类型别名：

|  |  |  |
| --- | --- | --- |
| **类型别名** | **map** | **set** |
| key\_type | 关键字类型 | 关键字类型 |
| mapped\_type | 关联类型 | **不存在该类型** |
| value\_type | pair<const key\_type, const mapped\_type> | key\_type |

### 5.2.1.关联容器迭代器

当解引用一个关联容器迭代器时，得到的是一个类型为容器的value\_type的值的引用。

对于set而言，value\_type就是key\_type。虽然提供了iterator和const\_iterator两种迭代器类型，但是set的迭代器只能访问其值，不能改变。效果都是const的。

map和set都支持begin和end操作，一样可以利用这一组迭代器遍历关联容器。

一般来说，**关联容器不使用泛型算法**。原因如下：

* 关联容器的关键字是const的，因此一些需要修改或者重排容器的算法，无法使用在关联容器上；
* 只读元素的算法，大多都要用到搜索操作。由于关联容器的关键字，并不能够用作快速查找的依据，因此效率上来说，使用泛型算法进行查找，对于关联容器而言是一个不好的选择。关联容器自己定义的find算法，效率会远超泛型算法find；

### 5.2.2.添加元素 -- insert

使用insert函数对关联容器执行插入元素的动作。

insert有两个版本，分别接受一对迭代器，或是一个初始化器列表。

#### 向map添加元素

向一个map执行insert操作时，元素类型必须是pair。通常对于要插入的数据，并没有一个现成的pair可以用，可以在insert的时候动态创建一个pair：

m.insert({a, 1});

m.insert(make\_pair(a, 1));

m.insert(pair<int, int>(a, 1));

m.insert(map<int, int>::value\_type(a, 1));

这几种方式都可以成功插入一个动态创建的pair。

#### 检测insert的返回值

对于不包含重复关键字的容器，返回一个pair：pair的first成员是一个迭代器，指向具有给定关键字的元素；second成员是一个bool变量，指出插入成功还是已经在容器中导致不能重复插入。

对于可以包含重复关键字的容器，insert必然成功，没有检测返回值的必要。

### 5.2.3.删除元素 -- erase

使用erase删除关联容器中的元素，支持以下三种形式:

|  |  |  |
| --- | --- | --- |
| **函数形式** | **说明** | **返回值** |
| c.erase(k) | 删除关键字为k的所有元素 | 返回一个size\_type的值，表明删除了多少个元素。如果不包含该元素，返回值为0； |
| c.erase(p) | 删除p这个迭代器指向的元素 | 返回指向p之后的元素的迭代器 |
| c.erase(b, e) | 删除b和e这对迭代器之间的元素 | 返回e |

### 5.2.4.map的下标操作 -- m[x]

set不支持下标操作，因为set不存在mapped\_type，只存在key\_type，使用关键字无法获取到一个与其关联的值，因此下标操作没有任何意义。

map和unordered\_map支持下标操作和一个对应的at函数，map下标接受一个索引（也就是一个key\_type类型的关键字的值），获取其值。

* 如果关键字存在，返回值；
* **如果关键字不存在，创建一个变量并插入到map中**。

由于下标运算符有可能执行插入新元素的动作，因此该操作不能对const类型的map使用。

### 5.2.5.访问元素 -- find

如果只是要查找一个元素是否存在于关联容器中：m.find(k)将会是最佳选择；

如果要统计一个元素在关联容器中出现了多少次，m.count(k)将会是更好的选择；

# 6.动态内存

## 6.1.动态内存与智能指针

C++中动态内存的申请和释放，使用的是new和delete操作；

由于动态内存的管理容易出现问题，因此标准库提供了一组智能指针来管理动态对象。智能指针的行为类似普通指针，重要的区别是他们负责自动释放所指向的对象，而不再需要自行释放。

标准库提供的智能指针有三种，都定义在memory头文件中：

1. shared\_ptr：允许多个指针指向同一个对象；
2. unique\_ptr：独占所指向的对象；
3. weak\_ptr：一种特殊的“弱引用”，指向shared\_ptr所管理的对象。

### 6.1.1.shared\_ptr类

与标准库的其他容器类似，shared\_ptr也是一个模板实现，定义时也需要传入具体指向的类型名称，例如：shared\_ptr<string > p1;就定义了一个shared\_ptr的p1对象，指向string类型的值。

智能指针的使用，与普通指针并无差别。解引用一个智能指针就返回它所指向的对象，例如：if(p1 && p1->empty()) {...}，判断条件中，先判断p1是否为空，不为空的情况下再判断其指向的对象(string类型)是否为空，如果为空就做某些操作。

#### make\_shared函数

最安全的分配和使用动态内存的方法，是使用make\_shared标准库函数。该函数在动态内存中分配一个对象并初始化它，返回指向该对象的shared\_ptr。例如：shared\_ptr<int> pA = make\_shared<int>(42);这里创建了一个指针，指向一个值为42的int变量，并将指针返回给了pA；

#### shared\_ptr的拷贝和赋值

在进行拷贝和赋值操作时，每个shared\_ptr都会记录有多少个其他的shared\_ptr指向相同的对象，例如：

auto p = make\_shared<int>(42); //p是一个shared\_ptr，指向的对象只有p这一个引用者；

auto q(p); //q和p指向同一个对象，这个对象现在有2个引用者了；

可以认为每个shared\_ptr都存在一个计数器，通常称之为引用计数(refrence count)。无论何时，拷贝一个shared\_ptr，都会给计数器加1；给shared\_ptr赋予一个新值或者shared\_ptr被销毁，计数器会减1。例如：

auto r = shared\_ptr<int>(43); //r指向的int型数据只有它一个引用者

r = q; //给r赋值，r不再指向上述的int型数据；

//这时候q指向的shared\_ptr，计数器会自加1

//而r指向的shared\_ptr，计数器会自减1

//由于r指向的int数据只有它一个引用者，因此自减之后已经没有引用者了，其shared\_ptr对象自动释放

### 6.1.2.直接管理内存

#### 使用new动态分配和初始化对象

默认情况下，动态分配的对象都是默认初始化的。这也就意味着，内置类型或者组合类型的对象的值是未定义的；而类类型对象将使用默认构造函数进行初始化，例如：

string \* p1 = new string; //初始化一个string，为空string；

int \* p2 = new int; //初始化一个int指针，未初始化；

也可以在动态分配空间的时候，直接使用初始化值执行初始化动作，例如：

string \* p3 = new string(“a”);

string \* p4 = new string(10, ‘c’);

int \* p5 = new int(3);

#### 内存耗尽

默认情况下，如果内存不足以分配出来new所申请的大小，会抛出bad\_alloc的异常。但可以通过改变使用new的方式来阻止抛出异常：

int \* p1 = new (nothrow) int; //这种方式下，如果分配失败，不会抛出异常，而是直接返回NULL。

bad\_alloc和nothrow，都定义在头文件new中。

#### 释放动态内存

通过delete将内存释放，归还给系统。

传递给delete的，必须是一个指针，且必须指向动态分配的内存。

释放一个非new分配的内存指针，或者一个指针释放多次，其后果都是未定义的。

delete之后，指针值就变成无效的了。虽然指针已经无效，但是很多系统中，指针仍然保存着这个动态地址的值。这就导致这个指针变成了空悬指针--dangling pointer。解决的办法，就是delete之后，将这个指针值用nullptr进行赋值。

#### 延伸：NULL和nullptr

nullptr是C++11中的关键字，表示空指针。

NULL是一个宏定义，在C和C++中定义不同：

* C中NULL是(void \*)0：#define NULL (void \*)0
* C++中则是整数0：#define NULL 0

nullptr是一个字面型常量，类型是std::nullptr\_t，空指针常数可以转换为任意类型的指针类型。

在C++中，void\* 不能转换为任意类型的指针，也就是说,int \* p = (void \*)a是错误的；但是int \*p = nullptr就是正确的。而且：

void fun(int i) {}

void fun(char \*p) {}

这两个函数是重载函数，在调用时，如果传入的是NULL，那么我们理论上想要调用的是后一个fun的实现，但C++却根据NULL的值选择了第一个给我们使用。

这时候如果使用的是nullptr，就能够正确的使用到第二个函数，因为它接收的参数是一个指针。

## 6.2.动态数组

为了使new分配一个对象数组，需要在类型名之后加一对方括号，在其中指明要分配的对象的数目。例如：int \*p = new int[40];将分配一个动态数组，大小是40，每个成员都是int型，p指向第一个int。

方括号中一定要是整形，但并不一定是常量。

在释放此类数组时，一定要方括号在前，例如：delete [] p;这样才能将整个动态数组的内存都释放掉，不造成内存泄漏。

# 7.拷贝控制

# 8.重载运算和类型转换

# 9.面向对象程序设计

高质量程序设计指南(C++/C语言)

# 1.入门(第4章)

⊙﹏⊙b汗，这一章哥好多都不知道……

## 1.1.基本概念

### 1.1.1.main函数

main函数应该返回int，但具体返回什么类型可以由实现来定义。不过所有实现版本都应该至少允许以下两种形式：

int main();

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

### 1.1.2.内部规范

在C语言中，所有函数不是局限于编译单元(文件作用域)的static函数，就是具有extern连接类型和global作用域的全局函数。这种情况下，除了两个不同编译单元中的static函数可以同名，其他都不可能出现同名的情况。所以C语言采用了简单的函数名称区分规则：仅在所有函数前添加“\_”，从唯一识别函数的作用来讲，实际上和不添加前缀没有差别。例如，main函数在连接时会被命名为\_main。

在C++中，允许用户在不同的作用域中定义同名的函数、类型、变量等，这些作用域不仅限于编译单元，还包括class、struct、union、namespace等，甚至在同一个作用域中也定义同名的函数(重载函数)。这种情况下编译器如果依旧只增加前缀“\_”，毫无疑问将出现同名函数等。

因此，C++的编译器都需要进行“Name-Mangling”(名字修饰 / 名字改编)的动作，会将所属作用域的名称(class、namespace等)及重载函数的参数信息(参数类型和个数等)作为修饰，由此作为其内部名称。

由于C++标准没有规定Name-Mangling的规则，所以各个编译器的连接器不能兼容。

# 2.C++/C编译预处理(第9章)

## 2.1.文件包含

为了避免同一个编译单元、包含同一个头文件的内容超过一次(这将导致**类型重复定义的错误**)，需要在头文件中使用**内部包含卫哨**。

例如：

#ifndef \_\_TEST\_H\_\_

#define \_\_TEST\_H\_\_

…

#endif

## 2.2.宏定义

宏分为带参数的宏和不带参数的宏。

宏定义具有文件作用域，不论宏定义出现在文件的哪个地方(函数体内、类型内部、名字空间内部等)，在它后面的任何地方都可以引用宏。

* 宏定义不是C++/C语句，不需要以“；”结尾；
* 宏定义可以嵌套：

#define PI 3.14

#define PI\_2 (PI\*2)

* 宏不可以被调试，因为宏不进入符号表(符号表是编译器创建的，而编译时宏已经消失了)；即使宏替换之后出现了语法错误，编译器也会将错误定位到源程序中，而不是具体的某个宏定义中；
* 带参数的宏，宏体和参数应分别用“()”括起来，例如：

#define SQUARE(x) ((x)\*(x))

* 不在宏的参数列表中使用增量或减量运算符，例如“int n = 5; SQUARE(n++);”，得到的结果取决于编译器的不同而不同。针对复合表达式中子表达式的顺序，不同的编译器可能有不同的执行标准。
* 如果需要公布某个宏，那么将该宏定义放在头文件中；否则放在实现文件中；
* 给宏添加注释时应使用块注释(/\*\*/)，而不使用行注释(//)。有些编译器可能会把行注释也理解成宏的一部分。

### 2.2.1.带参数的宏

* **不转义**

#define PI 3.14

#define PI\_2 (PI\*2)

#define SQUARE(x, x) ( (x)\*(x) )

* **转义成字符串**

使用“#”将宏参数转换成字符串，简单说就是可以将输入的参数左右分别加上“”””，使其以字符串的形式输出。

#define TEXT(str) #str

printf(TEXT(vck)); //输出将是“vck”这个字符串；

#define CHECK(EXP) do{if(EXP) fprintf(stderr, “Error : ” #EXP “\n”);}while(0)

CHECK(divider == 0)在编译时将被替换为：

do{if(divider == 0) fprintf(stderr, “Error : ” “divider == 0” “\n”);}while(0);

* **贴合**

使用“##”可以将两个宏参数贴合在一起，并不在乎前后两个参数是否是输入参数，只要是参数即可。

#define CONS(a, b) int(a##e##b)

printf(“%d”, CONS(2,3)); //输出将是2000。因为转换后是int(2e3)，2e3代表的是2\*103；

* **变参宏**

使用“…”可以定义一个变参宏，“…”必须出现在最后一个参数的位置，例如：

#define myPrint\_0(format,…) fprintf(stderr, format, ##\_\_VA\_ARGS\_\_)

#define myPrint\_1(format,args…) fprintf(stderr, format, args)

但是这种变参宏，只有C99标准才支持；而且如myPrint\_0所示的，会有多余的逗号出现，必须使用“##”加以限制；

## 2.3.条件编译

#if #elif #endif

#ifdef #ifndef #endif

## 2.4.预定义符号常量

|  |  |
| --- | --- |
| 符号常量 | 含义 |
| \_\_FILE\_\_ | 源文件名字 |
| \_\_LINE\_\_ | 引用该符号的代码行号 |
| \_\_DATE\_\_ | 引用该符号的源文件被编译的日期(字符串) |
| \_\_TIME\_\_ | 引用该符号的源文件被编译的时间(字符串) |
| \_\_TIMESTAMP\_\_ | 引用该符号的源文件被编译的日期和时间(字符串) |
|  |  |

# 3.C++面向对象程序设计方法概述

## 3.1.动态特性

绝大多数情况下，程序的功能是在编译的时候就确定的，称之为静态特性；反之，如果是在运行时才确定的，称之为动态特性。

C++虚函数、抽象基类、动态绑定(Dynamic bingding)和多态(Polymorphism)构成了出色的动态特性。

### 3.1.1.虚函数

虚函数的声明方法是在函数原型之前加上关键字virtual。

假定基类Shape，派生类Circle、Rectangle、Ellipse等，每个派生类都能绘制属于自己的图形。但是希望每个子类使用同样的绘制接口，那么就最好在Shape中定义接口函数Draw()，并让程序在运行时动态的确定应该使用哪个派生类的Draw()函数。

基类Shape中定义虚函数Draw，并在派生类中重新定义Draw，并使其绘制属于自己的形状，这种方法叫做**覆盖(override)**。

一旦类的一个函数声明为虚函数，那么派生类的对应函数也自动成为虚函数，这样一级级的传递下去。虽然如此，为了增加程序的清晰性，应该为每一级派生层次中，将其显示的声明为虚函数(virtual关键字)。

### 3.1.2.抽象基类

如果将基类的虚函数声明为纯虚函数，那么该类就被定义为了抽象基类。纯虚函数是在声明时将其函数“初始化”为0的函数，例如：virtual void Draw(void) = 0;

将一个函数初始化为0，意味着函数的地址是0，这就告诉编译器：不要为该函数编址，从而阻止该类的实例化。在C++中，**只有虚函数**才能被初始化为0.

抽象基类的主要用途是“接口和实现分离”：不仅要把数据成员(信息)隐藏起来，还可以把实现隐藏起来，只留一些接口给外部调用。

## 3.2.C++对象模型

TODO，本部分内容，当前看的一知半解。不抄书，看懂了在做记录。

# 4.对象的初始化、拷贝和析构

每个类只有一个析构函数，但可以有多个构造函数：默认构造函数，拷贝构造函数，其他构造函数；也可以有多个赋值函数：默认赋值函数，拷贝赋值函数，其他赋值函数。如果程序员不显示的定义这些函数，编译器将自动创建public inline的函数，但一旦类中包含指针、类有多层继承关系等现象，默认函数按位赋值的缺点将导致程序出现不可预期的错误。

## 4.1.为什么需要构造和析构函数

当给一个对象分配好原始空间的时候，这个对象就已经创建起来了。但此时的内存，没有任何有意义的值被分配，直接使用极大可能会出现问题。因此，创建起原始的内存空间后，需要为这个对象做初始化，使其内存空间有意义。

初始化不同于赋值：

* 初始化指的是对象创建的同时，使用初值直接填充对象的内存空间，因此不存在数据类型转换等中间过程，也不存在临时变量；
* 赋值是对象创建后任何时刻都可以调用的函数，由于它调用的是“=”运算符，因此可能需要进行类型转换，也就会产生临时变量；

C++对象可以使用构造函数执行初始化动作：构造函数是对象创建时自动调用的第一个成员函数，也是为每个对象仅调用一次的成员函数。

所以构造函数的作用就是：当对象的内存分配完成后，把对象从原始状态转变成良好的、可用的内存状态。

## 4.2.构造函数的成员初始化列表

构造函数初始化列表的使用规则如下：

* 如果类存在继承关系，派生类可以直接在其初始化列表里调用基类的特定构造函数；
* 类的非静态const数据成员和引用成员，只能在初始化列表里初始化，因为他们只存在初始化语义，不存在赋值语义；
* 类的数据成员的初始化可以采用初始化列表和函数体内赋值两种，其效率并不相同：
  + 如果数据成员是内部数据类型，如int char等，那么两种赋值方式的效率基本相同；
  + 如果是自定义数据类型，如其他类，那么初始化列表赋值只需要调用该类的拷贝构造函数即可；但构造函数体内赋值的方式需要先为成员变量创建对象，在调用该对象的赋值函数，才能完成赋值动作。显然前者的效率更高。

## 4.3.构造函数和赋值函数的重载

构造函数分三类：默认构造函数，拷贝构造函数，其他构造函数；

默认构造函数是这样的构造函数：或者没有参数，或者所有的参数都有默认值；

拷贝构造函数是这样的构造函数：第一个参数为本类对象的引用、const引用、volatile引用或const volatile引用，且没有其他参数，或其他参数都有默认值；

要注意，拷贝构造函数的第一个参数，**一定是引用**，而不能是对象值！

如果没有显示的定义默认构造函数，却定义了带参数的其他构造函数，那么后者的存在就会阻止编译器产生前者！导致的结果，就是这个类没有默认构造函数。

## 4.4.如何实现派生类的基本函数

基类的构造函数、析构函数、赋值函数都不能被派生类继承。如果类之间存在继承关系，在编写上述函数时需要注意如下几点：

1. 派生类的构造函数，应在其初始化列表里显示的调用基类的构造函数；
2. 如果基类是多态类，那么必须把基类的析构函数定义成虚函数！这样可以实现动态绑定，否则很可能造成内存泄漏。
3. 在编写派生类的赋值函数时，要对基类的数据成员重新赋值，这可以调用基类的赋值函数来实现；

# 5.C++函数的高级特征

## 5.1.成员函数的重载、覆盖和隐藏

成员函数被重载的特征是：

* 具有相同的作用域(位于同一个类中)；
* 函数名字相同；
* 参数类型、顺序或数目不同(包括const和非const参数的区别)；
* Virtual关键字可有可无；

成员函数被覆盖的特征是：

* 不同的作用域(分别位于基类和派生类中)；
* 函数名称完全相同；
* 函数参数列表完全相同；
* 基类函数必须声明为virtual函数；

隐藏函数指的是派生类的成员函数遮蔽了同名的基类函数，特征是：

* 派生类的函数与基类的函数同名，但是参数有差异。此时，不论有无virtual关键字，基类的函数在派生类中都将被隐藏；
* 派生类的函数名称和参数列表，都与基类的相同，但是基类没有virtual关键字。这时候，基类的函数将被隐藏；

## 5.2.参数的默认值

参数默认值的使用规则是：

* 把参数默认值放在函数声明中，而不是定义中；
* 如果函数存在多个参数，那么参数只能**从后向前**依次默认！

## 5.3.运算符重载

在C++中，使用关键字operator加上运算符来表示函数，称作运算符重载函数。例如，复数相加的函数，可以定义为：

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

如果运算符被重载为全局函数，那么只有一个参数的运算符叫做一元运算符，有两个参数的是二元运算符；

如果运算符被重载为类的成员函数，那么一元运算符没有参数(++和—的后置版本例外)，二元运算符有一个参数，因为对象自己成为了左侧的参数；

### 5.3.1.运算符重载的特殊性

1. 如果重载为成员函数，那么this对象发起对它的调用；
2. 如果重载为全局函数，那么第一个参数发起对它的调用；
3. 禁止用户自定义出运算符集合中不存在的运算符！
4. 除了函数调用运算符”()”外，其他运算符重载函数不能有默认参数值；

### 5.3.2.不能重载的运算符

1. 不能重载“.”；
2. 不能重载反引用类成员指针“.\*”；
3. 不能重载作用域解析运算符“::”；
4. 不能重载三元运算符“条件？A：B”；
5. 不能重载“sizeof”和“typeid”；
6. 不能重载“#”和“##”等预处理操作符；
7. 不能重载C++的新式类型转换运算符：static\_cast<>; dynamic\_cast<>;const\_cast<>;reinterpret\_cast<>；

### 5.3.4.重载++和—

当为一个类型重载“++”或“--”的前置版本时，不需要参数；而当为其重载后置版本时，需要一个int类型的参数作为标志，也就是哑元。

例如，前置版本：Complex operator++() {}

后置版本：Complex operator++(int) {}

当“++”和“--”应用于基本数据类型时，前置和后置效率并无明显差距；但当应用于用户自定义类型时，前置版本的效率将比后置版本高出很多。

## 5.4.函数内联

内联函数的目的是提高函数的执行效率，因为其省去了参数压栈、保存现场等工作。

与宏不同的是，内联函数是可以调试的。在程序的debug版本里，内联函数就是一个普通函数，因此可以调试；而在release版本里，内联函数才真正被内联进去，不可调试。

需要明确的是，inline关键字必须放在函数定义体之前才会生效，仅仅只是放在函数声明之前是不会生效的；如果声明时没有使用inline，只在定义时使用了inline，那么同样会按照内联函数处理。

所以说，内联是一种“用于实现的关键字”，而不是“用于声明的关键字”。

以下情况不应该使用内联：

1. 函数的代码过长，内联后将导致可执行程序的体积过大；
2. 函数体内有循环或其他复杂的控制结构，那么函数体的执行时间巨大，内联并无疑义；

## 5.5.Const成员函数

Const成员函数的定义，需要将关键字放在函数末尾，例如：int GetCount() const;

Const成员函数：

* 不能修改成员变量的值；
* 不能调用其它非const成员函数；

# 6.内存管理