目录

[C基础 1](#_Toc75958669)

[数组与字符 1](#_Toc75958670)

[String 2](#_Toc75958671)

[结构与共用 2](#_Toc75958672)

[枚举 4](#_Toc75958673)

[指针与数组 5](#_Toc75958674)

[创建动态结构 5](#_Toc75958675)

[存储类型 6](#_Toc75958676)

[综合类型 7](#_Toc75958677)

[数组的替代品 8](#_Toc75958678)

[基于范围的for循环 9](#_Toc75958679)

[嵌套与二维数组 10](#_Toc75958680)

[字符函数库 10](#_Toc75958681)

[Swith与枚举 12](#_Toc75958682)

[读取数字循环 13](#_Toc75958683)

[函数 14](#_Toc75958684)

[结构函数 15](#_Toc75958685)

[结构函数指针 16](#_Toc75958686)

[函数和string对象 17](#_Toc75958687)

[函数与array对象 18](#_Toc75958688)

[递归 19](#_Toc75958689)

[函数指针 20](#_Toc75958690)

[指针与字符 21](#_Toc75958691)

[函数指针深究 22](#_Toc75958692)

[文件输入 23](#_Toc75958693)

[文件输出 25](#_Toc75958694)

[内联函数 26](#_Toc75958695)

[引用变量 27](#_Toc75958696)

[结构引用 29](#_Toc75958697)

[类与引用 31](#_Toc75958698)

[对象继承与引用 32](#_Toc75958699)

[默认参数 34](#_Toc75958700)

[函数重载 35](#_Toc75958701)

[函数模板 37](#_Toc75958702)

[内存与名称空间 42](#_Toc75958703)

[对象和类 44](#_Toc75958704)

[类应用 45](#_Toc75958705)

[类和动态内存分配 46](#_Toc75958706)

[类设计 49](#_Toc75958707)

[类继承 50](#_Toc75958708)

[代码重用 54](#_Toc75958709)

[类模板 56](#_Toc75958710)

[队列 60](#_Toc75958711)

[友元及异常 63](#_Toc75958712)

[String类与标准模板 69](#_Toc75958713)

[文件操作 80](#_Toc75958714)

[C++新标准 91](#_Toc75958715)

## C基础

### 数组与字符

# include <iostream>

int main1()

{

using namespace std;

const int ArSize = 20;

char name[ArSize];

char dessert[ArSize];

int year;

//cin >> year;//这里结束后有回车符会被getline()采集到，误认为之后的输入是空行

(cin >> year).get();//不同输入类型的解决之道

cout << "Enter your name:\n";

//cin >> name;//解决方案，采用cin.getline(),回车键来确定该行是否结束，会删除换行符

//cin.get(name,ArSize);

//cin.get();//get()可以读取下一字符，及时是换行符

//cin.get(dessert, ArSize);//get(参数)函数会保留换行符，因此在第二个变量未输入时就认为已经换行结束

cin.get(name, ArSize).get();//cin.get(name, ArSize).get()=cin.get(name,ArSize);cin.get();

cout << "Enter your favourate dessert:\n";

//cin >> dessert;

//cin.getline(dessert, ArSize);

cin.getline(dessert, ArSize).get();

//当cin.get()或cin.getline()读取空行时，可以用cin.clear()恢复输入

//cin.get(name, ArSize)会返回一个cin对象，随后被用来调用get(),与cin.getline(name, ArSize).get(dessert, ArSize)效果一样

cout << "I have some delicious " << dessert;

cout << "for you," << name << ".\n";//在键盘输入时，程序将（name输入）空格视为第一个字符串结束(自动加空值字符\0)，并将后面的单词读入第二个输入

system("pause");

//cin.get();

//getchar();

//system("pause");

return 0;

}

### String

# include <iostream>

# include <string>//这里不能用cstring,不然string输出用s.c\_str()

int main2()

{

using namespace std;

char c1[20] = "jaguar";

char c2[20];

string str1 = "panther";

string str2;

string str3;

//cout << "Enter a kind of feline: ";

//cin >> c2;

//cout << "Enter another kind feline: ";

//cin >> str2;

str2 = str1;

str3 += str2;

strcpy\_s(c2, c1);//字符数组操作

strcat\_s(c1,"my");

int a =strlen(c1);

int b = str1.size();

cout << "Display the arry of char and sring...\n";

cout << "c1: " << c1<<endl << "c2: " << c2 << endl;

cout << "str1: " << str1<<"\n" << "str2: " << str2 << endl;

cout << "str3: " << str3 << endl;

cout << "The length of c1 is: " << a << endl;

cout << "The length of str1 is: " << b << endl;

system("pause");

return 0;

}

### 结构与共用

# include <iostream>

using namespace std;

struct inflatable//关键字与结构名 \*\*\*\*\*\*\*\*\*结构声明

{

char name[20];

float volume;

double price;

}//这里可以直接写结构体的实例变量

;

int main3()

{

inflatable guest = //\*\*\*\*\*\*\*结构体实例化，guest是inflatable 结构类型的一个变量

{

"Glorios Gloria",

1.88,

29.99

};

inflatable pal = //\*\*\*\*\*\*\*结构体实例化，pal是inflatable 结构类型的另一个变量

{

"Audiacious Arthur",

3.12,

32.99

};

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*结构数组\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

inflatable gifts[100];//创建某种结构体的多个实例化对象

inflatable al{ "Arica Dira", 12.56, 19.88 };

//初始化结构体时，与数组一样，利用“，”分隔

//

cout << "Expand your guest list with " << guest.name;

cout << " and " << pal.name ;

cout << " and " << al.name << "!\n";

cout << "You can have both for $";

cout << guest.price + pal.price+al.price << "!\n";

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*共用体\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//struct widget

//{

// char brand[20];

// int type;

// union id

// {

// long id\_num;

// char id\_char[20];

// } id\_val;//创建共用体类型及变量

//};//创建结构体类型

//widget price ;//实例化结构体

//if (price.type == 1)

//

// cin >> price.id\_val.id\_num;

// else

// cin >> price.id\_val.id\_char;

system("pause");

return 0;

}

### 枚举

# include <iostream>

using namespace std;

int main4()

{

enum spectrum{red,orange,yellow};//对应0-2

spectrum brand;//实例化变量

brand = red;//只有赋值运算，不能算数运算

enum bigstep{ first,second=100,third};//0,100,101

enum { zero, null, one, num = 1};//0,0,1,1

bigstep a = bigstep(50);//在范围内，合法

cout << a << endl;

cout << red << endl;

cout << third << endl;

system("pause");

return 0;

}

### 指针与数组

# include <iostream>

using namespace std;

int main5()

{

int nights = 1001;

int \*pt = new int;//allocate space for an int from heap or free store

\*pt = nights;//store a value there

cout << "nights value= ";

cout << nights << ": location = " << &nights << endl;

cout << "int ";

cout << "value = " << \*pt<< ": location = " << pt << endl;

//首先申请了int的内存，并复制nights的值给这块内存

delete pt;//释放内存

int noon=100;

int\* p;

p = &noon;//指针一定要给地址，不能直接\*p = noon

cout << "noon value= ";

cout << noon << ": location = " << &noon << endl;

cout << "int ";

cout << "value = " << \*p << ": location = " << p<< endl;

//\*\*\*\*\*\*\*\*\*\*\*数组的动态联编\*\*\*\*\*\*\*\*\*\*\*\*\*

int size;

cin >> size;

int \*pz = new int[size];//申请 size个int类型的内存，并返回首地址

\*pz = 5;//赋值5给第一个元素，其余随机

cout << pz[0] << pz[1] << pz[2];//\*(pz+k)=pz[k]

delete[] pz;//释放内存

system("pause");

return 0;

}

### 创建动态结构

# include <iostream>

struct inflable

{

char name[20];

float volume;

double price;

};

int main7()

{

using namespace std;

inflable \*ps = new inflable;//申请一块用于存储结构体的内存并返回首地址

cout << "Enter name of inflable item: ";

cin.get(ps->name, 20);//访问结构体成员的方法一

cout << "Enter volume in cubic feet: ";

cin >> (\*ps).volume;//访问结构体成员的方法二，（\*ps）=结构体名

cout << "Enter price:$";

cin >> ps->price;

cout << "Name: " << (\*ps).name << endl;

cout << "Volume: " << ps->volume << " cubic feet \n";

cout << "Price :$" << ps->price << endl;

delete ps;

system("pause");

return 0;

}

### 存储类型

# include <iostream>

# include <cstring>

using namespace std;

//该函数返回一个指向字符串的指针，将字符串读入一个大型的临时数组中，用

//new创建一个刚好可以存储它的内存，返回该内存块的指针，节省大量的内存空间

char \* getname();//函数声明

int main8()

{

char \*name;

name = getname();

cout << name << " at " << (int\*)name << "\n";

delete [] name;

name = getname();//重复利用之前的内存

cout << name << " at " << (int\*)name << "\n";

delete[] name;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*自动存储、静态存储、动态存储\*\*\*\*\*\*\*\*\*\*\*\*8

//自动存储：在函数内部定义的常规变量，在调用函数时产生，调用结束自动销毁，后进先出

//静态存储：在整个程序执行期间都存在，static

//动态存储： 管理一个内存池，称为自由存储空间或堆，与前两种变量的内存是独立的，若没有及时 delete会造成内存泄漏，即使包含

//指针的内存被释放，但内存池的仍然存在，而且无法访问，要避免出现这种现象。

system("pause");

return 0;

}

char \* getname()

{

char temp[80];

cout << "Enter last name: ";

cin >> temp;

char \*pn = new char[strlen(temp) + 1];

strcpy(pn, temp);

return pn;

}

### 综合类型

# include <iostream>

struct antarctica\_years\_end

{

int year;

/\*其他类型的数据\*/

};

/\*

int a[5]

int \*p=a;

a的类型是int[5] 数组

&a的类型是int(\*)[5] 指针——指向int[5]数组的指针

&a[0]的类型是int\* 指针——指向int类型的指针

a[k]=\*（p+k）

\*/

int main9()

{

antarctica\_years\_end s01, s02, s03;

s01.year = 1998;

antarctica\_years\_end \*pa = &s02;//结构体要取地址

pa->year = 1999;

antarctica\_years\_end trio[3];//数组名等价于指向数组首元素的指针

trio[0].year = 2003;

std::cout << trio->year << std::endl;

const antarctica\_years\_end \*arp[3] = { &s01, &s02, &s03 };

std::cout << arp[1]->year << std::endl;

const antarctica\_years\_end \*\*ppa = arp;//ppa指向arp,arp指向s01,s02,s03

auto ppb = arp;

std::cout <<"ppb: "<< ppb << std::endl;

std::cout << "arp: " << arp << std::endl;

std::cout << (\*ppa)->year << std::endl;//\*ppa=arp

std::cout << (\*(ppb + 1))->year <<std:: endl;//\*（ppb+1）=arp[1]即&s02

system("pause");

return 0;

}

### 数组的替代品

# include <iostream>

# include <vector>

# include <array>

using namespace std;

int main10()

{

//\*\*\*\*\*\*\*\*\*模板类vector\*\*\*\*\*\*\*\*

vector<int> vi;//创建一个vector<int>对象vi

int n;

cin >> n;

vector<double>vd(n);//vector<double>对象vd

//\*\*\*\*\*\*\*\*\*模板类array\*\*\*\*\*\*\*\*\*\*

array <int, 5> ai;

array <double, 4> ad = { 1.2, 2.1, 3.4, 4.3 };

//比较数组，array,vector

/\*

1、array、数组存储在栈中，vector存在自由存储区或堆中

2、array,vector可以整体复制，数组不行

3、下标为负，表示越界，可以用.at()降低犯错概率

\*/

double a1[4] = { 1.2, 2.4, 3.6, 4.8 };

double a6[4];

//a6 = a1;这里不行

vector <double> a2(4);

vector <double> a5(4);

a2[0] = 1.0 / 3.0;

a2[1] = 1.0 / 5.0;

a2[2] = 1.0 / 7.0;

a2[3] = 1.0 / 9.0;

a5 = a2;

array<double, 4>a3 = { 3.14, 2.72, 1.62, 1.4 };

array<double, 4>a4;

a4 = a3;//数组不能这样写

cout << "a1[2]: " << a1[2] << " at " << &a1[2] << endl;

cout << "a2[2]: " << a2[2] << " at " << &a2[2] << endl;

cout << "a3[2]: " << a3[2] << " at " << &a3[2] << endl;

cout << "a4[2]: " << a4[2] << " at " << &a4[2] << endl;

cout << "a5[2]: " << a5[2] << " at " << &a5[2] << endl;

a1[-2] = 20.2;

cout << "a1[-2]: " << a1[-2] << " at " << &a1[-2] << endl;

cout << "a3[2]: " << a3[2] << " at " << &a3[2] << endl;

cout << "a4[2]: " << a4[2] << " at " << &a4[2] << endl;

system("pause");

return 0;

}

### 基于范围的for循环

# include <iostream>

using namespace std;

int main11()

{

double prices[5] = { 4.99, 10.99, 6.87, 7.95, 8.49 };

for (double x : prices)

{

cout << x << endl;

}

//哨兵字符

char ch;

int count = 0;

cout << "Enter characters; enter # to quit:\n";

//cin >> ch;

cin.get(ch);

while (ch != '#')

{

cout.put(ch);

++count;

//cin >> ch;//继续输入,不包括回车空格，制表符

cin.get(ch);//

}

cout << endl << count << "characters read\n";

system("pause");

return 0;

}

### 嵌套与二维数组

# include <iostream>

using namespace std;

int main12()

{

const int Cities = 5;

const int Years = 4;

const char \*cities[Cities] =

{

"TIANJIN", "BEIJIN", "WUHAN", "TISNDHUI", "XIAN"

};

int maxtemps[Years][Cities] =

{

{23,12,34,43,67},

{34,54,21,54,23},

{54,21,36,47,48},

{24,13,13,35,36}

};

cout << "Maximum temperatures for 2008-2011\n\n";

for (int city = 0; city < Cities; ++city)

{

cout << cities[city] << ":\t";

for (int year = 0; year < Years; ++year)

{

cout << maxtemps[year][city] << "\t";

}

cout << endl;

}

system("pause");

return 0;

}

### 字符函数库

# include <iostream>

# include <cctype>

using namespace std;

int main13()

{

cout << "Enter text for analysis,and type @"

"to terminate input.\n";

char ch;

int whitescape = 0;

int digits = 0;

int chars = 0;

int punct = 0;

int others = 0;

cin.get(ch);

while (ch != '@')

{

if (isalpha(ch))//是字母

chars++;

else if (isspace(ch))//是标准空白符

whitescape++;

else if (isdigit(ch))//是数字

digits++;

else if (ispunct(ch))//是标点

punct++;

else

others++;

cin.get(ch);//得到下一个字符

}

cout << chars << "letters, "

<< digits << "digits, "

<< punct << "punctuations, "

<< others << "others.\n";

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*“？:”\*\*\*\*\*\*\*\*\*\*\*

const char x[2][20] = { "Jason ", " at your service\n" };

const char \*y = " Quillstone ";

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

{

cout << ((i < 2) ? !i ? x[i] : y : x[1]);

}

/\*

i=0,(i<2)?x[0]:x[1]--->x[0]

i=1,(i<2)?y:x[1]--->y

i=0,(i<2)?y:x[1]--->x[1]

\*/

/\*

isalnum():是字母或数字

iscntrl():是控制字符

isgragh():是除空格外的打印字符

islower():是小写

isprint():是打印字符

isupper():是大写

isxdigit():是十六进制

tolower():若是大写，则返回小写，否则返回参数

toupper():若是小写，则返回大写，否则返回参数

\*/

system("pause");

return 0;

}

### Swith与枚举

# include <iostream>

using namespace std;

int main14()

{

enum {red,orange,yellow,green,blue,violet,indigo};

cout << "Enter color code(0-6): ";

int code;

cin >> code;

while (code >= red && code <= indigo)

{

switch (code)

{

case red:cout << "Her lip were red.\n"; break;

case orange:cout << "Her hair was orange.\n"; break;

case yellow:cout << "Her shoes were yellow.\n"; break;

case green:cout << "Her nails were green.\n"; break;

case blue:cout << "Her sweatsuit was blue.\n"; break;

case violet:cout << "Her eyes were violet.\n"; break;

case indigo:cout << "Her mood were indigo.\n"; break;

}

cout << "Enter color code(0-6): ";

cin >> code;

}

cout << "Bye\n";//只有越界才会输出

system("pause");

return 0;

}

### 读取数字循环

# include <iostream>

using namespace std;

const int Max = 5;

int main15()

{

//get\_data

double fish[Max];

cout << "Please enter the weight of your fish.\n";

cout << "You may enter up to "<<Max

<<"fish <q to terminate>.\n";

cout << "fish # 1:";

int i = 0;

while ((i < Max)&& cin >> fish[i])//这里输入鱼的重量，如果i>Max,直接结束循环

{

if (++i < Max)

cout << "\n" << "fish # " << i + 1 << ":" ;

}

//while循环结束3

//calculate average

double total = 0.0;

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

total += fish[i];

if (i == 0)

cout << "No fish\n";

else

cout << "total:" << total << endl;

cout << total / i << "= average weight of " << i << " fish\n";

cout << "Done.\n";

system("pause");

return 0;

}

### 函数

# include <iostream>

using namespace std;

//\*\*\*\*\*\*\*\*\*C++的返回值不能为数组，但可以是结构和对象，因此可以将数组作为结构、对象的组成来返回

//不指定参数列表时应该用...

//实参：argument,形参：parameter

/\*

处理函数常用的编写方式：

函数不修改数组：void fun(const double x[]);//a[]=\*a，指出a是指针

函数可以修改数组：void fun(double \*x);

指定数组区间：void fun(int\* begin,int\*end);//调用时给定数组区间地址

const int \*pt=&a; int \*pt=&a;//可以通过pt和p访问a的值，但不能通过p修改a，而且p的值可修改

int \*const pm,则pm的值也无法修改

禁止将const赋给非const指针，除非强制转换

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*函数与二维数组\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

arr=&arr[0],&arr=整个数组的首地址，arr[k]=\*(arr+k),k<n;

int data[3][4]=｛｛1,2,3,4｝，｛4,5,6,7｝，｛7,8,9,0｝｝;

int (\*data)[4]=int data[][4];//定义一个数组指针，data指向int [4]的数组

//int \*data[4];//定义一个指针数组，4个元素为指向int 的指针

data指向数组（它的元素是4个int组成的数组）的第一个元素，因此data[r]指向编号为r的元素，该元素为一个数组，故data[r]也是由4个int元素组成的数组的名称

data[r][c]是由4个int组成的数组的第一个元素：data[r][c]=\*（\*（data+r）+c）：\*（data+r）=data[r]

data[1]=\*(data+1)=｛4,5,6,7｝=arr[4]

data[1][3]=\*(data[1]+3)=\*(arr+3)=7

\*/

unsigned int c\_in\_str(const char \*str, char ch);//函数声明

int main18()

{

char mmm[15] = "minimum";

char \*wail = "ululate";//wail point to string

unsigned int ms = c\_in\_str(mmm, 'm');

unsigned int us = c\_in\_str(wail, 'u');

cout << ms << " m characters in " << mmm << endl;

cout << us << " u characters in " << wail << endl;

system("pause");

return 0;

}

unsigned int c\_in\_str(const char\* str, char ch)

{

unsigned int count = 0;

while (\*str)//quit when \*str is '\0'

{

if (\*str == ch)

count++;

str++;//move pointer to next char

}

return count;

}

### 结构函数

//直角坐标系转换为极坐标系

# include <iostream>

# include <cmath>

using namespace std;

struct polar

{

double diatance;

double angle;

};

struct rect

{

double x;

double y;

};

polar rect\_to\_polar(rect xypos);

void show\_polar(polar dapos);

int main19()

{

rect rplace;

polar pplace;

cout << "Enter the x and y values: ";

while (cin >> rplace.x >> rplace.y)//调用cin时将返回一个istream对象，cin是istream的一个对象，因此，多次调用抽取运算符>>得到的仍是istream值，判断输入是否符合类型返回true/false

{

pplace = rect\_to\_polar(rplace);

show\_polar(pplace);

cout << "Next two numbers(q to quit): ";//'q'不是double

}

cout << "Done.\n";

system("pause");

return 0;

}

polar rect\_to\_polar(rect xypos)

{

polar answer;

answer.diatance = sqrt(xypos.x\*xypos.x + xypos.y\*xypos.y);

answer.angle = atan2(xypos.y, xypos.x);

return answer;

}

void show\_polar(polar dapos)

{

const double Rad\_to\_deg = 57.29577951;//弧度转角度

cout << "distance= " << dapos.diatance;

cout << ", angle= " << dapos.angle\*Rad\_to\_deg;

cout << "degrees\n";

}

### 结构函数指针

//直角坐标系转换为极坐标系

# include <iostream>

# include <cmath>

using namespace std;

struct polar

{

double diatance;

double angle;

};

struct rect

{

double x;

double y;

};

void rect\_to\_polar(const rect \*pxy,polar \*pda);//没有返回值，直接在参数中定义可修改的变量

void show\_polar(const polar \*pda);

int main20()

{

rect rplace;

polar pplace;

cout << "Enter the x and y values: ";

while (cin >> rplace.x >> rplace.y)//调用cin时将返回一个istream对象，cin是istream的一个对象，因此，多次调用抽取运算符>>得到的仍是istream值，判断输入是否符合类型返回true/false

{

rect\_to\_polar(&rplace,&pplace);

show\_polar(&pplace);

cout << "Next two numbers(q to quit): ";//'q'不是double

}

cout << "Done.\n";

system("pause");

return 0;

}

void rect\_to\_polar(const rect \*pxy,polar \*pda)

{

pda->diatance = sqrt(pxy->x\*pxy->x + pxy->y\*pxy->y);

pda->angle = atan2(pxy->y, pxy->x);

}

void show\_polar(const polar \*pda)

{

const double Rad\_to\_deg = 57.29577951;//弧度转角度

cout << "distance= " << pda->diatance;

cout << ", angle= " << pda->angle\*Rad\_to\_deg;

cout << "degrees\n";

}

### 函数和string对象

# include <iostream>

# include <string>

using namespace std;

const int SIZE = 5;

void display(const string sa[], int n);

int main21()

{

string list[SIZE];//an array holding 5 string object

cout << "Enter your" << SIZE << "favorite astronomical sights:\n";

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

{

cout << i + 1 << ":";

getline(cin, list[i]);

}

cout << "Your list:\n";

display(list, SIZE);

system("pause");

return 0;

}

void display(const string sa[], int n)

{

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

{

cout << i + 1 << ":" << sa[i] << endl;

}

}

### 函数与array对象

# include <iostream>

# include <array>

# include <string>

//光标变成方块按insert键

const int Season = 4;

const std::array<std::string, Season>Snames =

{ "Spring", "Summer", "Fall", "Winter" };//function to modify array object

//Snames 是array对象的一个实例，即四个元素的字符串数组

void fill(std::array<double, Season>\*pa);//对象的地址传递 function that uses array object without modifying it

void show(std::array<double, Season>da);//对象的值传递

using namespace std;

int main22()

{

array<double, Season>expenses;

fill(&expenses);

show(expenses);

system("pause");

return 0;

}

void fill(std::array<double, Season>\*pa)

{

using namespace std;

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

{

cout << "Enter" << Snames[i] << "expenses: ";

cin >> (\*pa)[i];

}

}

void show(std::array<double, Season>da)

{

using namespace std;

double total = 0.0;

cout << "\nEXPENSES\n";

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

{

cout << Snames[i] << ": $" << da[i] << endl;

total += da[i];

}

cout << "Total Expenses: $" << total << endl;

}

### 递归

# include <iostream>

using namespace std;

const int Len = 66;

const int Divs = 6;

void subdivide(char ar[], int low, int high, int level);

int main23()

{

char ruler[Len];

int i;

for (i = 1; i < Len - 2; i++)

ruler[i] = ' ';

ruler[Len - 1] = '\0';

int max = Len - 2;

int min = 0;

ruler[min] = ruler[max] = '|';

std::cout << ruler << endl;

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

{

subdivide(ruler,min,max,i);

cout << ruler << endl;

for (int j = 1; j < Len - 2; j++)

ruler[j] = ' ';

}

system("pause");

return 0;

}

void subdivide(char ar[], int low, int high, int level)

{

if (level == 0)

return;

int mid = (high + low) / 2;

ar[mid] = '|';

subdivide(ar, low, mid, level - 1);

subdivide(ar, mid, high, level - 1);//调用一次level减一

}

### 函数指针

# include <iostream>

double betsy(int);//函数声明中可以省略标识符，但在定义函数时必须说明标识符

double pam(int);

void estimate(int lines, double(\*pf)(int));

using namespace std;

int main24()

{

//函数名即函数地址，函数名（）为返回值。

/\*

int \* pt (int)//定义一个函数，其返回值为int\*类型

int （\*p）(int)//定义一个函数指针p，指向int fun(int a);

\*/

int code;

cout << "How many lines of code do you need ? ";

cin >> code;

cout << "Here's Betsy's estimate:\n";

estimate(code, betsy);

cout << "Here's Pam's estimate:\n";

estimate(code, pam);

//调用函数时可以用\*函数名或直接使用函数名

system("pause");

return 0;

}

void estimate(int lines,double (\*pf)(int))

{

cout << lines << "lines will take ";

cout << (\*pf)(lines) << "hours\n";//调用函数

}

double betsy(int lns)

{

return 0.05\*lns;

}

double pam(int lns)

{

return 0.03\*lns + 0.0004\*lns\*lns;

}

### 回调函数

回调函数机制：

回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。  
1、定义一个函数（普通函数即可）；  
2、将此函数的地址注册给调用者；  
3、特定的事件或条件发生时，调用者使用函数指针调用回调函数。

### 指针与字符

# include <iostream>

# include <cstring>

int main6()

{

using namespace std;

char animal[20] = "bear";

const char \* bird = "wren";//"wren"=s[],bird=s

char \* ps;

cout << animal << " and ";

cout << bird << "\n";

//cout << ps << "\n";//这里由于ps未初始化会导致编译错误

cout << "Enter a kind of animal: ";

cin >> animal;

ps = animal;

cout << ps << "!\n";//直接输出fox,与使用animal一样，打印字符数组名等于打印字符串

cout << "Before uisng strcpy(): \n";

cout << animal << " at " << (int\*)animal << endl;//强制类型转换

cout << ps << " at " << (int\*)ps << endl;

ps = new char[strlen(animal) + 1];

strcpy(ps, animal);//接收的参数是字符数组的地址

cout << "After using strcpy(): \n";

cout << animal << " at " << (int\*)animal << endl;

cout << ps << " at " << (int\*)ps << endl;

delete [] ps;

system("pause");

return 0;

}

//char a[]={'a','a','a','\0'};

//str\_\_\_(const char \*,const char \*,int a,……)：包含在<cstring>中，形参均为字符指针

//std::string l="asb";//自动添加‘\0’

### 函数指针深究

# include <iostream>

using namespace std;

//various notation,same signatures

const double \* f1(const double ar[], int n);

const double \* f2(const double [], int );

const double \* f3(const double \*, int );

//只调用函数f1(const double ar[], int n)，则返回的是double 返回值的地址

typedef const double \*(\*p\_fun)(const double \*, int);//定义一种类型

int main25()

{

double av[] = { 1112.3, 1542.6, 2227.9 };

cout << &av[0] << endl;

const double \* (\*p1)(const double \*，int);

auto p2 = f2;

p\_fun p3 = f3;//类型的定义使得程序出错可能性减少

cout << "Using pointer to functions:\n";

cout << "Address Value\n";

cout << f1(av, 3) << ": " << \*f1(av, 3)<< endl;

const double \* (\*pa[3])(const double \*, int) = { f1, f2, f3 };

//auto 不能用于一个序列的初始化，但可以用于单值的初始化

auto pb = pa;

cout << "\nUsing an array of pointer to functions:\n";

cout << "Adress Value\n";

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

{

cout << pa[i](av, 3) << ":" << \*pa[i](av, 3) << endl;

}

cout << "\nUsing an array of pointer to functions:\n";

cout << "Adress Value\n";

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

{

cout << pb[i](av, 3) << ":" << \*pb[i](av, 3) << endl;

}

cout << "\nUsing an array of pointer to functions:\n";

cout << "Adress Value\n";

auto pc = &pa;//pc为函数的双指针

//const double \*(\*（\*pc）[3])(const double\*, int) = & pa;

cout << (\*pc)[0](av, 3) << ": " << \*(\*pc)[0](av, 3) << endl;

//(\*pc)[0](av, 3)为av 中double值的地址

const double \*(\*(\*pd)[3])(const double\*, int) = &pa;

//(\*pd)[3]:pd存储一维数组首地址

const double \*pdb = (\*pd)[1](av,3);//先取首地址为pa,然后向后偏移

cout << pdb << ": " << \*pdb << endl;

system("pause");

return 0;

}

const double \*f1(const double \*ar, int n)

{

return ar;

}

const double \* f2(const double ar[], int n)

{

return ar + 1;

}

const double \*f3(const double ar[], int n)

{

return ar + 2;

}

### 文件输入

# include <iostream>

# include <fstream>

using namespace std;

int main16()

{

char automobile[50];

int year;

double a\_price;

double b\_price;

ofstream outFile;//创建一个输出对象

outFile.open("carinfi.txt");//将对象与文件关联,并建立一个文本文件；

cout << "Enter the make and model of automobile: ";

cin.getline(automobile, 50);

cout << "Enter the model year: ";

cin >> year;

cout << "Enter the original asking price: ";

cin >> a\_price;

b\_price = 0.913\*a\_price;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*输出流控制\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

boolalpha 可以使用单词”true”和”false”进行输入/输出的布尔值.

oct 用八进制格式显示数值.

dec 用十进制格式显示数值.

hex 用十六进制格式显示数值.

left 输出调整为左对齐.

right 输出调整为右对齐.

scientific 用科学记数法显示浮点数.

fixed 用正常的记数方法显示浮点数(与科学计数法相对应).

showbase 输出时显示所有数值的基数.

showpoint 显示小数点和额外的零，即使不需要.

showpos 在非负数值前面显示”＋（正号）”.

skipws 当从一个流进行读取时，跳过空白字符(spaces, tabs, newlines).

unitbuf 在每次插入以后，清空缓冲区.

internal 将填充字符回到符号和数值之间.

uppercase 以大写的形式显示科学记数法中的”e”和十六进制格式的”x”.

\*/

//cout << fixed<<precision(2)

cout << fixed;//浮点数的输出以小数形式

cout.precision(2);//设置小数精度为后两位

cout.setf(ios\_base::showpoint);//显示小数点和额外的0

cout << "Make and model: " << automobile << endl;

cout << "Year: " << year << endl;

cout << "Was asking $" << a\_price << endl;

cout << "Now asking $" << b\_price << endl;

//创建的输出对象可以使用cout的任何方法

outFile << fixed;

outFile.precision(2);

outFile.setf(ios\_base::showpoint);

outFile << "Make and model: " << automobile << endl;

outFile << "Year: " << year << endl;

outFile << "Was asking $" << a\_price << endl;

outFile<< "Now asking $" << b\_price << endl;

outFile.close();//关闭文本文件

system("pause");

return 0;

}

### 文件输出

# include <iostream>

# include <fstream>//file I/O support

# include <cstdlib>//support for exit()

const int SIZE = 60;

using namespace std;

int main()

{

char filename[SIZE];

ifstream inFile;//creat a object for input

cout << "Enter name of data file: ";

cin.getline(filename, SIZE);

inFile.open(filename);//associate object with file

if (!inFile.is\_open())//fail to open file

{

cout << "Could not open the file " << filename << endl;

cout << "Program terminating.\n";

exit(EXIT\_FAILURE);

}

double value;

double sum = 0.0;

int count=0;

inFile >> value;

while (inFile.good())

{

++count;

sum += value;

inFile >> value;//按double 类型读取文本数据

}

if (inFile.eof())//程序读取文件不应该超过EOF

cout << "End of file reached.\n";

else if (inFile.fail())//检查类型是否匹配

cout << "Input terminate by data mis match.\n";

else

cout << "Input terminate for unknow reason.\n";

if (count == 0)

cout << "No data processed.\n";

else

{

cout << "Items read: " << count << endl;

cout << "Sum: " << sum << endl;

cout << "Average: " << sum / count << endl;

}

inFile.close();

system("pause");

return 0;

}

### 内联函数

# include <iostream>

using namespace std;

//内联函数的编译代码与其他程序的代码“内联”了，运行速度快，但占用的内存也比较多

inline double square(double x){ return x\*x; }

int main1()

{

double a, b;

double c = 13.0;

a = square(5.0);

b = square(4.5 + 7.5);

cout << "a= " << a << ",b= " << b << endl;

cout << "c= " << c << endl;

cout << ",c squared = " << square(c++) << endl;

cout << "Now c= " << c << endl;

system("pause");

return 0;

}

### 引用变量

# include <iostream>

using namespace std;

//交换两个变量的值意味着值传递将不可以

#pragma region not\_fun

void swapr(int &a, int &b);

void swapp(int \*p, int \*q);

void swapv(int a, int b);//不可行

double cube(double a);

double refcube(double &ra);//按引用传递会改变实参的值

#pragma endregion

//当引用的对象是const而且类型不正确或不是左值时，会生成临时变量

/\*

应尽可能使用const引用

1、使用const可以避免无意中修改数据的编程错误

2、使用const 使函数能够处理const和非const实参，否则将只接受非const数据

3、使用const引用使函数能够正确生成并使用临时变量

\*/

//C++11新增了右值引用

double &&reff = sqrt(36.0);

int main2()

{

//与\*运算符一样，在定义或声明变量时，&表示引用变量，表示两个变量共用同一地址和值

#pragma region MyRegion

int rats = 101;

int &rodents = rats;

cout << "rats= " << rats << endl;

cout << ",rodents= " << rodents << endl;

rodents++;

cout << "rats= " << rats << endl;

cout << ",rodents= " << rodents << endl;

cout << "address rats= " << &rats << endl;

cout << ",address rodents= " << &rodents << endl;

int bunnies = 50;

rodents = bunnies;//引用与原变量的值变了，但地址未变

cout << "bunnies= " << bunnies << endl;

cout << "rats= " << rats << endl;

cout << ",rodents= " << rodents << endl;

cout << "address bunnies= " << &bunnies << endl;

cout << "address rats= " << &rats << endl;

cout << "address rodents= " << &rodents << endl;

#pragma endregion

#pragma region swap

int wallet1 = 300;

int wallet2 = 320;

cout << "wallet1=$ "<<wallet1;

cout << "wallet2=$ " << wallet2<<endl;

cout << "Using references to swap contents:\n";

swapr(wallet1, wallet2);

cout << "wallet1=$ " << wallet1;

cout << "wallet2=$ " << wallet2 << endl;

cout << "Using references to swap contents agin:\n";

swapp(&wallet1, &wallet2);

cout << "wallet1=$ " << wallet1;

cout << "wallet2=$ " << wallet2 << endl;

cout << "Trying to use passing by value:\n";

swapv(wallet1, wallet2);

cout << "wallet1=$ " << wallet1;

cout << "wallet2=$ " << wallet2 << endl;

#pragma endregion

#pragma region cube

double x = 3.0;

cout << cube(x);

cout << "= cube of " << x << endl;

cout << refcube(x);

cout << "= cube of " << x << endl;

#pragma endregion

system("pause");

return 0;

}

void swapr(int &a, int &b)//按引用传递

{

int temp=a;

a = b;

b = temp;

}

void swapp(int \*p, int \*q)//按地址传递

{

int temp = \*p;

\*p = \*q;

\*q = temp;

}

void swapv(int a, int b)//按值传递，相当于复制了原实参的值

{

int temp = a;

a = b;

b = temp;

}

double cube(double a)

{

a \*= a\*a;

return a;

}

double refcube(double &ra)

{

ra \*= ra\*ra;

return ra;

}

### 结构引用

# include <iostream>

# include <string>

struct free\_throws

{

std::string name ;//string name出错,未使用名称空间

int made;

int attempts;

float percent;

};

void display(const free\_throws &ft);//结构引用声明

void set\_pc(free\_throws &ft);

free\_throws &accumulate(free\_throws & target, const free\_throws &source);//返回值为结构体引用的函数

//返回应用应避免返回函数终止时不再存在的内存单元的引用，如临时变量

using namespace std;

int main3()

{

//initial

free\_throws one = { "Ifelsa Branch", 13, 14 };

free\_throws two = { "Andor Branch", 19, 13 };

free\_throws three = { "Minnies Branch", 15, 14 };

free\_throws five = { "Whilyy Branch", 18, 14 };

free\_throws four = { "Long Branch", 17, 13};

free\_throws team = { "Throed Branch", 0, 0 };

//no initial

free\_throws dup;

//display(one);

set\_pc(one);

display(one);

accumulate(team, one);

display(team);

cout << "名称不同" << endl;

display(accumulate(team, two));

accumulate(accumulate(team, three), four);

display(team);

dup = accumulate(team, five);

cout << "display team:"<<endl;

display(team);

set\_pc(four);

accumulate(dup, five);

cout << "display dup after ill-advised assignment:" << endl;

display(dup);

system("pause");

return 0;

}

void display(const free\_throws &ft)

{

using std::cout;

cout << "Name: " << ft.name << endl;

cout << "Made: " << ft.made << endl;

cout << "Attempts: " << ft.attempts << endl;

cout << "Percent: " << ft.percent << endl;

}

void set\_pc(free\_throws &ft)

{

if (ft.attempts != 0)

ft.percent = 100.0f\*float(ft.made) / float(ft.attempts);

else

ft.percent = 0;

}

free\_throws &accumulate(free\_throws & target, const free\_throws &source)//返回值为结构体的函数

{

target.attempts += source.attempts;

target.made += source.made;

set\_pc(target);

return target;

}

### 类与引用

# include <iostream>

# include <string>

using namespace std;

string version1(const string &s1, const string &s2);

const string &version2(string &s1, const string &s2);

const string &version3(string &s1, const string &s2);

int main4()

{

string input;

string copy;

string result;

cout << "Enter a string: ";

getline(cin, input);

copy = input;

cout << "Your string as entered: " << input << endl;

result = version1(input, "\*\*\*");

cout << "Your string enhanced: " << result << endl;

cout << "Your original string: " << input << endl;

result = version2(input, "\*\*\*");

cout << "Your string enhanced: " << result << endl;

cout << "Your original string: " << input << endl;

cout << "Resetting original string .\n";

input = copy;

result = version3(input, "@@@");//错误的设计

cout << "Your string enhanced: " << result << endl;

cout << "Your original string: " << input << endl;

system("pause");

return 0;

}

string version1(const string &s1, const string &s2)

{

string temp;

temp = s2 + s1 + s2;

return temp;

}

const string &version2( string &s1, const string &s2)

{

s1 = s2 + s1 + s2;

return s1;

}

const string &version3(string &s1, const string &s2)//程序试图引用已经释放的内存

{

string temp;

temp = s2 + s1 + s2;

return temp;

}

### 对象继承与引用

/\*

使用引用参数的几种情况：

1、可以修改调用参数中的数据对象

2、通过传递引用而不是数据对象，可以提高程序的运行速度（较大的结构和类对象）

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*值传递与引用传递和指针传递\*\*\*\*\*\*\*\*

/\*

1、如果数据对象很小，如内置数据类型或小型结构，则按值传递

2、数据对象是数组，使用const指针

3、数据对象是较大的结构，使用const 指针或 const 引用，可以节省复制所需要的时间和空间

4、数据对象是类对象，使用const引用

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*修改调用函数数据的函数\*\*\*\*\*\*\*\*\*

/\*

1、数据对象是内置数据类型，使用指针

2、数据对象是数组，只能使用指针

3、数据对象是结构，则使用引用或指针

4、数据对象是类对象，使用引用

\*/

# include <iostream>

# include <fstream>

# include<cstdlib>

using namespace std;

void file\_it(ostream &os, double fo, const double fe[], int n);

const int LIMIT = 5;

//ofstream是派生类，ostream是基类，因此ofstream的对象（fout）或引用对象(os)可以使用基类的方法

//派生类继承了基类的方法，基类引用可以指向派生类对象

int main5()

{

ofstream fout;

const char \*fn = "ep\_data.txt";

fout.open(fn);

if (!fout.is\_open())

{

cout << "Can't open" << fn << ". Bye.\n";

exit(EXIT\_FAILURE);

}

double objective;

cout << "Enter the focal length of your""telescope objective in mm: ";

cin >> objective;

double eps[LIMIT];

cout << "Enter the focal length ,in mm,of " << LIMIT << "eyepieces:\n";

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

{

cout << "Eyepieces # " << i + 1 << ": ";

cin >> eps[i];

}

file\_it(fout, objective, eps, LIMIT);//写入文件

file\_it(cout, objective, eps, LIMIT);//显示

cout << "Done\n";

system("pause");

return 0;

}

//参数os（其类型是 ostream &）即 既可以指向ostream对象（cout）,又可以指向ofstream对象（fout）

void file\_it(ostream &os, double fo, const double fe[], int n)

{

ios\_base::fmtflags initial;//存储使用函数之前的信息的数据类型,没有的话变成科学计数法表示

initial = os.setf(ios\_base::fixed);//设置输出流为定点输出格式

os.precision(0);

os << "Focal length of objective: " << fo << "mm\n";//1800无小数输出

os.setf(ios::showpoint);

os.precision(1);//一位小数

//os.width(12);

os << "f.1. eyepieces";

//os.width(15);

os << "magnification" << endl;

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

{

os.width(12);

os << fe[i];

os.width(15);

os << int(fo / fe[i] + 0.5) << endl;

}

os.setf(initial);//还原数据类型

}

/\*

ios\_base::flags，读取/设置流的格式。

ios\_base::setf，设置流的格式，与原有格式合并。

ios\_base::unsetf，根据参数mask，清除流的格式的某些位（bit）。

ios\_base::precision，读取 / 设置显示浮点数时的精度。

ios\_base::width，读取 / 设定流的输出数据的显示宽度。

\*/

### 默认参数

# include <iostream>

using namespace std;

const int ArSize = 80;

char \*left1(const char \* str, int n = 1);//只能从右向左设置默认参数，在调用函数时可以省略该参数

int main6()

{

char sample[ArSize];

cout << "Enter a string: \n";

cin.get(sample, ArSize);

char \*ps = left1(sample, 4);//函数返回值为指针，类型匹配

cout << ps << endl;

delete[] ps;//释放内存,因为在函数调用时使用了new char[]

ps = left1(sample);

cout << ps << endl;

delete[] ps;

system("pause");

return 0;

}

char \*left1(const char \*str, int n)

{

if (n < 0)

{

n = 0;

}

char \*p = new char[n + 1];

int i;

for (i = 0; i < n&&str[i]; i++)

{

p[i] = str[i];

}//继续保持i,读取完n个非空字符之后在后面赋‘\0’

while (i <= n)

p[i++] = '\0';

return p;

}

### 函数重载

# include <iostream>

using namespace std;

unsigned long left(unsigned long num, unsigned ct);

char \*left(const char \*str, int n = 1);

//函数多态（重载）允许可以使用多个同名函数，完成相同功能，使用不同参数

//返回类型可以不同，但特征标（参数）必须不同

//非const 可以赋给const，但反之不行

//同一项目中不能定义相同的函数

/\*

左值指既能够出现在等号左边，也能出现在等号右边的变量；右值则是只能出现在等号右边的变量

左值是可寻址的变量，有持久性；

右值一般是不可寻址的常量，或在表达式求值过程中创建的无名临时对象，短暂性的。

int x = 6; // x是左值，6是右值

int &y = x; // 左值引用，y引用x

int &z1 = x \* 6; // 错误，x\*6是一个右值

const int &z2 = x \* 6; // 正确，可以将一个const引用绑定到一个右值

int &&z3 = x \* 6; // 正确，右值引用

int &&z4 = x; // 错误，x是一个左值

\*/

int main7()

{

char \*trip = "Hawaii";

unsigned long n = 12345678;

int i;

char \*temp;

for (i = 1; i < 10; i++)

{

cout << left(n, i) << endl;

temp = left(trip, i);

cout << temp << endl;

delete[] temp;

}

system("pause");

return 0;

}

unsigned long left(unsigned long num, unsigned ct)

{

unsigned digits = 1;

unsigned long n = num;

if (ct == 0 || num == 0)

return 0;

while (n /= 10)

digits++;//统计数的位数

if (digits > ct)

{

ct = digits - ct;

while (ct--)

num /= 10;//由低位向高位求解

return num;

}

else

return num;//要求截取的位数大于给定的数字位数时，输出原数字

}

char \* left(const char \*str, int n)//截取字符串

{

if (n < 0)

n = 0;

char\*p = new char[n + 1];

int i;

for (i = 0; i < n&&str[i]; i++)

p[i] = str[i];

while (i <= n)

{

p[i++] = '\0';

}

return p;

}

### 函数模板

# include <iostream>

using namespace std;

//模板定义

/\*

template <class AnyType>

void Swap(AnyType &a, AnyType &b)

{

AnyType temp;

temp = a;

a = b;

b = temp;

}

\*/

/\*

函数重载选择优劣：

1、完全匹配，但常规函数优先于模板

2、提升转换（char/short-->int ,float转化为double）变长

3、标准转换（int--char，long---double）

4、自定义转换

\*/

template <typename T>//or class T

void Swap(T &a, T &b);

template<typename T>//overload template

void Swap(T \*a, T \*b, int n);

void Show(int a[]);

const int Lim = 8;

//显式具体化

struct job//可以整体交换赋值

{

char name[40];

double salary;

int floor;

};

template<>void Swap(job &j1, job &j2);//具体化-----显式实例化

void Show(job &j);//函数重载

template <class T>//自定义选择

T lesser(T a, T b)

{

return a < b ? a : b;

}

int lesser(int a, int b)

{

a = a < 0 ? -a : a;

b = b < 0 ? -b : b;

return a < b ? a : b;

}

template <class T1,class T2>

void ft(T1 x, T2 y)

{

decltype(x + y) z = x + y;//已知x与y类型，z与(x+y)类型一致

}

template <class T1, class T2>

auto gt(T1 x, T2 y)->decltype(x + y)

{

return x + y;

}

int main()

{

cout.precision(2);

cout.setf(ios::fixed, ios::floatfield);

//第一个参数指出要设置哪些位，第二个参数指出要清除哪些位。

int i = 10;

int j = 20;

cout << "i,j= " << i << "," << j << endl;

cout << "Using compiler generated int swapper:\n";

Swap(i, j);

cout << "Now i,j= " << i << "，" << j << endl;

double x = 24.5;

double y = 81.7;

cout << "x,y= " << x<< "," << y << endl;

cout << "Using compiler generated int swapper:\n";

Swap(x, y);//具体化-----隐式实例化

cout << "Now x,y= " << x << "，" << y << endl;

int d1[Lim] = { 0, 7, 0, 4, 1, 7, 7, 6 };

int d2[Lim] = { 0, 7, 2, 0, 1, 9, 6, 9 };

cout << "Oriiginal arrays:\n";

Show(d1);

Show(d2);

Swap(d1, d2, Lim);

cout << "Swapped arrays:\n";

Show(d1);

Show(d2);

# pragma region special

job sue = { "Sidny Yaffee", 73000.60, 7 };

job sidney = { "Sidny Yaffee", 70806.72, 9 };

cout << "Before job swapping:\n";

Show(sue);

Show(sidney);

Swap(sue, sidney);

cout << "After job swapping:\n";

Show(sue);

Show(sidney);

#pragma endregion

#pragma region choice

int m = 20;

int n = -30;

double x1 = 15.5;

double y1 = 25.9;

cout << lesser(m, n) << endl;//非模板

cout << lesser(x1, y1) << endl;//模板

cout << lesser<>(m, n) << endl;//模板

cout << lesser<int>(x1, y1) << endl;//模板---显式实例化

#pragma endregion

#pragma region lambda

//lambda函数表达式（匿名函数）：auto 函数名=[capture](parameters)->return-type{body}

//直接通过函数名调用

//[捕获从句]（形参列表）-> 类型说明符｛函数体｝指明返回类型

//[捕获从句]（形参列表)｛函数体｝未指明返回类型，自动识别

/\*

[] // 沒有定义任何变量。使用未定义变量会引发错误。

[x, &y] // x以传值方式传入（默认），y以引用方式传入。

[&] // 任何被使用到的外部变量都隐式地以引用方式加以引用。

[=] // 任何被使用到的外部变量都隐式地以传值方式加以引用。

[&, x] // x显式地以传值方式加以引用。其余变量以引用方式加以引用。

[=, &z] // z显式地以引用方式加以引用。其余变量以传值方式加以引用。

[this]：通过引用捕获当前对象（其实是复制指针）；

[\*this]：通过传值方式捕获当前对象；

\*/

//lambda表达式无法修改通过复制形式捕捉的变量，因为函数调用运算符的重载方法是const属性的。

//左侧返回值、参数，右侧所执行的功能，即函数体

//无参数，无返回值：

auto m1 = [](){ cout << "Hello World" << endl; };

getchar();

m1;

int x2 = 10;

auto add\_x = [x2](int a) mutable { x2 \*= 2; return a + x2; }; // 复制捕捉x

std::cout << add\_x(10) << endl; // 输出 30

#pragma endregion

system("pause");

return 0;

}

template <typename T>

void Swap(T &a, T &b)

{

T temp;

temp = a;

a = b;

b = temp;

}

template <typename T>

void Swap(T a[], T b[],int n)

{

T temp;

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

{

temp = a[i];

a[i] = b[i];

b[i] = temp;

}

}

void Show(int a[])

{

cout << a[0] << a[1] << "/";

cout << a[2] << a[3] << "/";

for (int i = 4; i < Lim; i++)

cout << a[i];

cout << endl;

}

//模板的局限性:数据类型本身不符合的运算，如数组的整体赋值（结构可以整体赋值），结构的比较等等

template<>void Swap<job>(job &j1, job &j2)

{

double t1;

int t2;

t1 = j1.salary;

j1.salary = j2.salary;

j2.salary = t1;

t2 = j1.floor;

j1.floor = j2.floor;

j2.floor = t2;

}

void Show(job &j)

{

cout << j.name << ": $" << j.salary

<< "on floor " << j.floor << endl;

}

/\*

标志 功能

boolalpha 可以使用单词"true"和"false"进行输入/输出的布尔值.

oct 用八进制格式显示数值.

dec 用十进制格式显示数值.

hex 用十六进制格式显示数值.

left 输出调整为左对齐.

right 输出调整为右对齐.

scientific 用科学记数法显示浮点数.

fixed 用正常的记数方法显示浮点数(与科学计数法相对应).

showbase 输出时显示所有数值的基数.

showpoint 显示小数点和额外的零，即使不需要.

showpos 在非负数值前面显示"＋（正号）".

skipws 当从一个流进行读取时，跳过空白字符(spaces, tabs, newlines).

unitbuf 在每次插入以后，清空缓冲区.

internal 将填充字符回到符号和数值之间.

uppercase 以大写的形式显示科学记数法中的"e"和十六进制格式的"x".

\*/

## 内存与名称空间

//通常，编译器使用三块独立的内存：一块用于静态变量，一块用于自动变量（栈），另一块用于动态存储（堆）

//\*\*\*\*\*\*\*\*\*\*\*\*存储持续性\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

自动存储持续性（栈）：函数中声明的一般变量，局部作用域，函数运行时存在，不会自动初始化

register（寄存器）：显式指出自动变量

注意auto在C++11后用于自动类型推断

静态存储持续性（固定的内存块）：函数定义外定义的变量或static定义的变量，程序运行过程中一直存在

外部链接性(函数外变量)：可在其他文件中访问

引用声明（不初始化）：extern,在多个文件中使用外部变量，则只需要定义一次，但在引用的文件中要引用声明

内部链接性（static）：只能在当前文件中访问

同一变量在同一文件中只能定义一次（单定义规则），但可以使用static定义静态外部变量，在该文件中静态变量将隐藏常规外部变量

无链接性(函数中static)：只能在当前函数或代码块中访问，适于再生，只初始化一次

局部变量与全局变量为同名时，局部变量将隐藏全局变量

线程存储持续性：thread\_local，生命周期与所属线程一样长

动态存储持续性：new分配的，自由存储或堆

\*/

//使变量不冲突的几种办法

/\*

1、静态 static 只在当前文件或函数中有效，并且会隐藏重名的外部变量

2、引用 extern （没重新定义）=原外部变量

extern （重新定义）--局部变量，隐藏原外部变量

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*说明符与限定符\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

register:指示寄存器存储

static：作用域为整个文件时，表示内部链接性；用于局部声明时，局部变量的存储持续性是静态的

extern:引用声明，其他地方定义的变量

thread\_local:变量的持续性与所属线程的持续性相同

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* cv限定符 \*\*\*\*\*

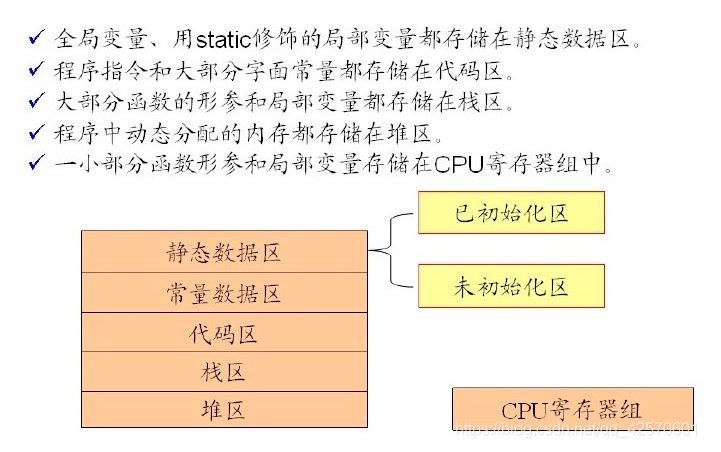
/\*

const:使全局变量的链接性变为内部性

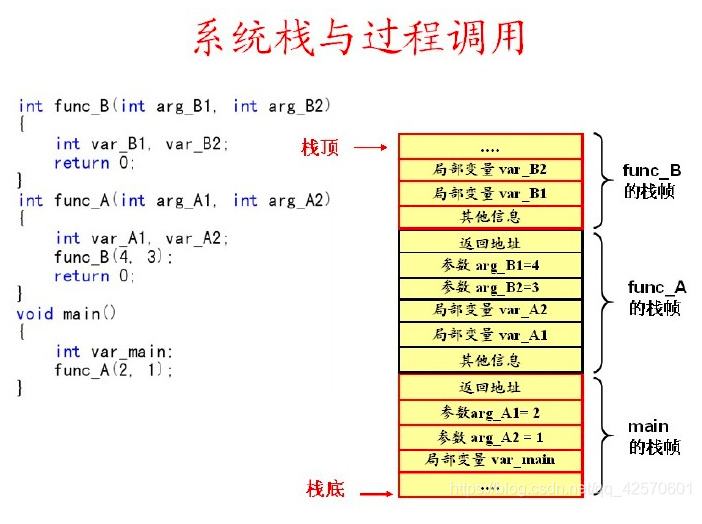
volatile（不稳定的）:代码即使没有对内存单元修改，其值也可能发生改变

mutual（成员限定符）：即使结构体或类为const，其某个成员也可以被修改

\*/



**堆指：**（满足堆性质的）优先队列的一种数据结构，第1个元素有最高的优先权；  
**栈指**：满足先进后出的性质的数学或数据结构。



## 对象和类

//OOP特性

/\*

1、抽象

2、封装和数据隐藏

3、多态

4、继承

5、代码的可重用性

\*/

//类方法可以访问private组件

//定义成员函数时，使用作用域解析运算符::标识函数所属的类

//类与结构体有很大的类型性，不过结构体默认访问是public，而类是private

//客户/服务器模型只允许修改类设计的实现细则，不能修改接口

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*类的构造函数与解析函数\*\*\*\*\*\*\*\*\*\*\*\*

/\*

构造函数与构析函数没有返回值

类的数据隐藏特性，使得其初始化与结构不同

构造函数（函数名一般为类名）的参数不是类的成员，而是类成员要赋的值，因此，参数名不能与类成员名相同。

一种常见的做法是使用m\_前缀或\_后缀。

隐式调用构造函数创建对象：Stock garment("anm",20,23)====》Stock garment=stock("anm",20,23)

对象生成在构造函数之后，所以不能通过对象来调用。

默认构造函数是在未提供显式初始值时，用来创建对象的构造函数。

定义默认构造函数的方式：

1、为已经创建的构造函数参数提供默认值

2、通过函数重载定义（无参数）

析构函数：对象过期时用于清理new产生的内存

\*/

//C++11支持列表形式初始化类

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*this指针\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//方法涉及两个对象

//this指针用来指向调用函数的对象（this指针被作为隐藏参数传递给方法）

//每个成员函数都可以有this指针

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*对象数组\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//利用默认构造函数声明对象数组，不用初始化定义

//利用一般构造函数直接创建和初始化对象数组

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*类作用域\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

访问类的成员必须通过对象借助（.）,(::),(->)来访问

为了使枚举中的枚举量不冲突，可以使用作用域内枚举

enum class(struct) egg{small,big};enum class(struct) pal{small,mid};

枚举底层类型改变：enum class:short pizza{small,mid,big};

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*抽象数据类型(ADT)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

栈的操作：

1、可以创建空栈

2、可将数据项添加到堆顶（压入）

3、可从栈顶删除数据项（弹出）

4、可查看栈是否填满

5、可查看栈是否为空

\*/

## 类应用

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*重载运算符\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*重载限制\*\*\*\*\*\*\*\*\*\*

/\*

1、重载后的运算符必须至少有一个操作数是用户自定义的

2、不能违反原来的句法规则

3、不能创建新的运算符

4、不能重载下列运算符

sizeof、:、\*、::、？：、typeid、const\_cast、dynamic\_cast、reinterpret\_cast、satic\_cast（强制类型转化符）；

5、下面的运算符只能通过成员函数重载

=、()、[]、->

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*友元\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

在重载二元运算符时用到，在类声明中需要声明

创建关键符：friend

cout<<x<<y:cout是ostream类的一个对象，而运算符<<的参数是ostream的对象引用，返回值也是对象（的引用），即返回值仍为ostream

因此cout<<x返回一个ostream对象cout引用，故可以用（cout<<x）<<y

友元在定义时不需要再用friend

\*/

/\*

一个矢量类：

srand(time(0)):随机数种子

定义类特定的常量可以采用

enum{a=14};

static const int a=14;

类中定义的枚举变量可以通过：：直接使用

\*/

/\*

强制类型转换符号重载：

operator typename();可隐式可显式

explicit operator typename();只能显式

int Stonewt::Stone\_to\_int(){return int(pounds+0.5)}只能显式

将加法定义为友元更容易适应类型转换，避免二义性

\*/

## 类和动态内存分配

//静态类成员有一个特点：无论创建了多少对象，

//只创建一个静态变量副本，即所有的对象共享一个静态成员

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*C风格字符串\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

假设要将字符串作为参数传递给函数，则表示字符串的方式有三种

1、char 数组

2、用引号括起来的字符串常量

3、被设为字符串的地址的char指针（char\*）

char ghost[15]="galloping";

char \*str="global";

int n1=strlen(ghost);

int n2=strlen(str)

int n3=strlen("my bad day");

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*类成员中有指针时\*\*\*\*\*\*\*\*\*\*

/\*

构造函数的参数为字符串指针时，字符串并不保存在对象中。字符串单独保存在堆中，对象仅保存了指出要去哪里查找字符串的信息。

每个对象的成员应该是独立的，而不能共享堆中的数据，可以在初始化时将堆中数据通过指针及new拷贝过来，

在对象以后的操作可能会改变其中的成员，相当于副本的作用，增加程序安全性。

必须定义复制构造函数，计数更新，深度复制

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*特殊成员函数\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

默认构造函数，如果没有定义构造函数，只能有一个，否则会有二义性

默认析构函数，如果没有定义，在创建对象（构造函数创建和复制构造函数创建时）后，对象过期时会调用，因此

需要增加复制构造函数被调用的计数，即提供计数可更新的显式复制构造函数和深度复制，使每个对象都有自己的字符串（地址）而不是

其他对象的引用

复制构造函数(函数按值传递)，如果没有定义

如果有一个已经定义的对象motto，则下面两张方式会调用：

String1 ditto(motto)和String1 metto=motto;

逐个复制非静态成员

成员函数返回对象时（返回局部对象时不能使用引用返回），调用复制构造函数，但返回引用不会

赋值运算符，如果没有定义，经常与复制构造函数的出现相关

由于目标对象（左值）可能引用了以前分配的数据，应该用delete []释放

避免将对象赋值给自身

返回引用

地址运算符，如果没有定义，返回调用对象的地址即this指针的值

移动构造函数

移动赋值运算符

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*标准输入流\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

使用 cin 从标准输入读取数据时，通常用到的方法有 cin>>、cin.get()，cin.getline()。

当我们从键盘输入字符串的时候需要敲一下回车键才能够将这个字符串送入到缓冲区中，

那么敲入的这个回车键（\r）会被转换为一个换行符（\n），这个换行符也会被存储在 cin

的缓冲区中并且被当成一个字符来计算！比如我们在键盘上敲下了 123456 这个字符串，

然后敲一下回车键（\r）将这个字符串送入了缓冲区中，那么此时缓冲区中的字节个数是 7 ，

而不是 6。

cin>>:

当 cin>> 从缓冲区中读取数据时，若缓冲区中第一个字符是空格、tab或换行这些分隔符时，

cin>> 会将其忽略并清除，继续读取下一个字符，若缓冲区为空，则继续等待。但是如果读取成功，

字符后面的分隔符是残留在缓冲区的，cin>> 不做处理。

不想略过空白字符，那就使用 noskipws 流控制。比如 cin>>noskipws>>input;

cin.get(char)

cin.get() 从输入缓冲区读取单个字符时不忽略分隔符，直接将其读取

cin.get() 的返回值是 int 类型，成功则返回读取字符的 ASCII 码值

cin.get(char\*s arsize )

遇到换行符时结束读取，但是不对换行符进行处理，换行符仍然残留在输入缓冲区。

将截取输入缓存区中的前n位

cin.get(char\* arsize delim)

指定读取的行数

cin.getline()

cin.getline 读取一行字符时，默认遇到 ‘\n’ 时终止，并且将 ‘\n’ 直接从输入缓冲区中删除掉，不会影响下面的输入处理。

//\*\*\*\*\*\*\*\*\*\*\*\*\*状态与清除\*\*\*\*\*\*\*\*\*\*\*

goodbit(0x0)：无错误

eofbit(0x1)：已到达文件尾

failbit(0x2)：非致命的输入/输出错误，可挽回

badbit(0x4)：致命的输入/输出错误，无法挽回

与这些条件状态对应的就是设置、读取和判断条件状态的流对象的成员函数：

s.eof()：若流 s 的 eofbit 置位，则返回 true

s.fail()：若流 s 的 failbit 置位，则返回 true

s.bad()：若流 s 的 badbit 置位，则返回 true

s.good()：若流 s 的 goodbit 置位，则返回 true

s.clear(flags)：清空当前状态, 然后把状态设置为 flags，返回 void

s.setstate(flags)：不清空当前状态，设置给定的状态 flags，返回 void

s.rdstate()：返回流 s 的当前条件状态，返回值类型为 ios\_base::iostate

getline()

getline() 利用 cin 可以从标准输入设备键盘读取一行，当遇到如下三种情况会结束读操作：

文件结束\遇到行分隔符/输入达到最大限度。

getline() 函数的参数使用了 string 字符串.

gets()

gets() 是 C 中的库函数，在头文件 <stdio.h> 申明，从标准输入设备读字符串，可以无限读取，不会判断上限，以回车或者文件结束符 EOF（ 即 -1） 结束

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*使用new初始化对象\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

Class\_name \*pclass= new Class\_name（value）;

会调用构造函数：Class\_name(Type\_name);

new用于为对象分配内存（1个或多个），也就是为保存字符串指针和len成员分配，这里是一个对象的内存，故用delete不是delete[]

调用的构造函数为存储字符串分配内存

用new创建的对象在显式使用delete时才会调用析构函数

其他new:

Type\_name \*p=new Type\_name [N];

必须用delete[]，堆

定位new:定位 new，就是预先分配一块内存，再将 new 出来的对象放入此内存块中的方法。通俗点说，就是在指定内存块中产生新对象

Type\_name \*p=new (Arr\_name) Type\_name [N];

Type\_name \*p=new (Arr\_name) Type\_name

不需要delete，每次调用使用相同内存块

对于定位new创建的对象，应该以与创建顺序相反的顺序进行删除

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*指针与对象\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

1、常规声明对象的指针

String1 \*glamour;

2、可以将指针初始化为指向已有的对象

String1 \*first=& sayings[0];

3、可以使用new来初始化指针，这将创建一个新对象

String1 \*pveg= new String("cabbag heads home");

1.为对象分配内存：str:--,len:---

2.调用类构造函数：

为"cabbag heads home"分配空间，将"cabbag heads home"复制到分配的内存中

将"cabbag heads home"地址给str,将长度给len，更新静态变量num\_strings

3.创建pveg变量

4.将新对象的地址给pveg

4、对类使用new将调用相应的构造函数来初始化创建的对象

String1 \*gl=new String1;

5、可以使用->运算符通过指针访问类方法；

sayings[i].length()=shortest->length();

6、可以使用解除引用运算符\*来获得对象

sayings[i]=\*first;

\*/

## 类设计

类设计回顾：

1、构造函数，在创建对象时，通过该函数给类的私有成员赋值

默认构造函数：在没有定义构造函数时

如果在派生类的构造函数的成员初始化列表中没有显式调用基类构造函数

则编译器将使用基类的默认构造函数来构造派生类对象的基类部分。

一般（显式）构造函数，需要给类的私有成员赋初值或者动态内存分配时

派生类的对象在创建之前要首先执行基类的构造函数

复制构造函数（对象过期会调用析构函数）：接受其所属类的对象或对象引用作为参数

1、将新对象初始化为一个同类对象；

2、按值将对象传递给函数

3、函数按值返回对象

4、编译器生成临时对象

临时对象通常产生于以下4种情况：类型装换、按值传递、按值返回、对象定义

2、赋值运算符

默认的赋值运算符主要用于处理同类对象之间的赋值，修改已有的对象的值

在使用之前考虑清除左值对象的内容。

3、析构函数：一定要定义显式的虚构函数释放构造函数使用new分配的所有内存

对于基类，即使它不需要析构函数，也应该提供虚析构函数（virtual）动态联编

4、转换（自动向上，禁止向下）：

格式：operator typename();//可隐式可显式

定义构造函数时（一个未知参数作隐式转换）：BaseClass(const char\*)//convert char\* to baseclass

explicit Star(const char\*)//禁止隐式转换，允许显式转换

5、按值传递与按引用传递（按引用传递）

6、按值返回和按引用返回

一般按引用返回

在返回临时对象时只能按值返回

7、使用const

8、友元（friend）：并非类成员，不能继承，可以访问类的私有成员。

如果希望通过派生类的友元函数使用基类的友元函数，可以通过强制类型转换将派生类的引用或指针转换为基类引用或指针，借此来调用基类的友元函数

9、虚方法（virtual）：希望派生类中能够重新定义基类函数或者使用动态联编（多态性）

10、ABC接口： class expendclass :public baseclass{……纯虚方法（=0）};

纯虚函数在ABC类中声明，没有函数实现，一般在派生类中实现（函数体）

11、protected是相对于派生类的一个概念，派生类的成员可以直接访问基类的保护成员，但不能访问基类的私有成员。

由protected声明的成员称为“受保护的成员”，或简称“保护成员”。从类的用户角度来看，保护成员等价于私有成员。但有一点与私有成员不同，保护成员可以被派生类的成员函数引用。

12、static

## 类继承

//类继承：

/\*

类库是以源代码的方式提供的，可以修改，但C++提供了一种比修改代码更好的方法来扩展和修改类，称为类继承。

1、在已有类的基础上添加功能

2、给类添加数据

3、可以修改类方法

表示方法：

派生类：public 基类

{

……

};

派生类对象存储了基类的数据成员（派生类继承了基类的实现）

派生对象可以使用基类的方法（派生类继承了基类的接口）

1、派生类需要自己的构造函数

2、派生类可以根据需要添加额外的数据成员和成员函数。

class RatedPlayer: public TableTennisPlayer

{

private:

usigned int rating;

public:

RatedPlayer(unsigned int r=0,const string & fn="none",const string &ln="none",bool ht=false);//派生类构造函数，不能直接访问基类的数据成员，只能通过基类的成员函数。

RatedPlayer(unsigned intr,const TableTennisPlayer &tp);//通过基类的复制构造函数赋值

unsigned int Rating() const{return rating;}

void ResetRating(unsigned int r){rating=r;}

}

基类对象在程序进入派生类构造函数之前被创建。

RatedPlayer::RetaedPlayer(unsigned int r,const string &fn,const string &ln,bool ht):TableTennisPlayer(fn,ln,ht)

{

rating=r;

}

或

RatedPlayer(unsigned intr,const TableTennisPlayer &tp):TableTennisPlayer(tp)

{

rating=r;

}

1、首先创建基类对象

2、派生类构造函数应通过初始化列表将基类信息传递给基类构造函数。

3、派生类构造函数应初始化派生类新增的数据成员

注意：创建派生类对象时，程序首先调用基类构造函数，然后再调用派生类构造函数。基类构造函数负责初始化继承的数据成员；

派生类构造函数主要用于初始化新增的数据成员。派生类的构造函数总是调用一个基类构造函数。可以使用初始化器列表语法指明要使用的基类构造函数，

否则将使用相应的默认构造（复制构造函数）函数。派生类对象过期时，程序将首先调用派生类析构函数，然后再调用基类析构函数。

derived::derived(type1 x,type2 y):base (x,y)

{

……

}

\*/

//基类和派生类之间的特殊关系

/\*

1、派生类对象可以使用基类的方法

2、基类指针可以在不进行显式类型转换的情况下指向派生类对象。

但不可以将基类对象和地址赋给派生类指针和引用。基类包含派生类

对于形参为指向基类的指针的函数，可以使用基类对象的地址或派生类对象的地址作为实参。

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*继承：is-a关系\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

is-a====is a kind of：A----->B（单向）

常见的关系：

has a:午餐有水果，但水果不一定是午餐

is like(像) a:律师像鲨鱼，但律师并不是鲨鱼

is implemented as a(作为……来实现):数组可以实现栈，但是栈不是数组

uses a：计算机可以使用打印机，但从计算机派生打印机无意义

多态公有继承：同一个方法在派生类和基类中的行为是不同的

1、在派生类中重新定义基类的方法

2、使用虚方法（程序将根据引用或指针的对象选择方法）

\*/



//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*联编\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*8

/\*

将源代码中的函数调用解释为执行特定的函数代码块被称为函数名联编

在编译过程中进行联编被称为静态联编，又称为早期联编

由于虚函数的引进使得编译器的选择变得复杂，编译器必须生成能够在程序运行时选择正确的虚方法的代码称为动态联编，又称为晚期联编

静态联编比动态联编更合理也更高效。

1、如果要在派生类中重新定义基类的方法，则设为虚方法，否则设非虚。

//指针与引用的兼容性

将派生类引用或指针转换为基类引用或指针被称为向上强制转换，是继承的特性。可以隐式转换，但是向下转换必须显式。

虚函数的形参分别为基类值，引用，指针时，基类对象和派生类对象调用虚函数时，只有指针和引用会发生隐式的向上转换呈现多态性

按值传递导致只将派生类对象的基类部分传递给虚函数。

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*虚函数\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

虚函数表：保存了指向所有基类函数与虚函数的地址数组的指针，在定义基类或者在派生类定义虚函数时才会更新

基类对象包含一个指针，该指针指向基类中所有虚函数的地址表（vtbl），派生类对象将包含一个指向独立地址表的指针。

如果派生类提供了虚函数的新定义，该虚函数表（派生对象的独立地址表）将保存新函数的地址，否则不更新。

使用虚函数的成本：

1、每个对象都将增大，增大量为存储地址的空间

2、对于每一个类，编译器将创建一个虚函数地址表

3、对于每个函数调用，都需要执行一项额外的操作，即到表中查找地址

注意：

1、在基类方法的声明中使用关键字virtual可以使该方法在基类以及所有的派生类中是虚的

2、如果使用指针或引用（对象的），程序将使用为对象类型定义的方法，而不使用为引用或指针类型定义的方法，即动态联编。

3、如果定义的类将被用作基类，则应将那些要在派生类中定义的类方法声明为虚的。

4、构造函数不能为虚，析构函数一般为虚，友元不能为虚。

5、(virtual)重新定义将隐藏方法，可以将返回值类型改为派生类对象引用（返回类型协变）

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*访问控制：protected\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

protected是相对于派生类的一个概念，派生类的成员可以直接访问基类的保护成员，但不能访问基类的私有成员。

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*抽象基类（ABC:abstrct base class）\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

至少使用一个纯虚的接口（=0），包含纯虚函数的类只能用作基类，它是集多个类的共性于一体，它不能创建对象，可以管理多个类

ABC=class1 交 class2

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*继承与动态内存分配\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

1、派生类不使用new

假设基类使用了动态内存分配，派生类未使用new，则不需要为派生类显式定义析构函数，复制构造函数和赋值运算符

2、派生类使用new

假设派生类使用了new（\*p）,必须为派生类显式定义析构函数，复制构造函数和赋值运算符

\*/

## 代码重用

//小范围/具象类（信息多）：大范围/抽象类（信息少）

//包含，组合或层次化：类的成员本身是另一个类的对象。

//has-a关系，即新的类包括另一个类的对象。

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*接口与实现\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

使用公有继承时，类可以继承接口，可能还有实现（基类的纯虚函数提供接口，但不提供实现），获得is--a关系的组成部分。

使用组合，类可以获得实现，但不能获得接口。不继承接口是has--a关系的组成部分。

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*valarray对象\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

double gpa[5]={3.1,4.5,6.7,7.8,3.3};

valarray<double>v1;//an array of double,size 0

valarray<int>v2(8);//an array of 8 int elements

valarray<int>v3(10,8)//an array of 8 int elements,each set to 10

valarray<double>v4(gpa,4);//an array of 8 int elements,initialed to the first 4 elements of gpa

valarray<double>v5={23,34,45,34,12};

该类的方法：

operator[]():访问各个元素

size():返回包含的元素个数

sum():返回元素的总和

max(),min()

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*被包含对象的接口\*\*\*\*\*\*\*\*8

/\*

被包含对象的接口不是公有的，但是可以用类方法访问

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*继承\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//继承类型主要决定基类的（公有成员和保护成员）在派生类的访问权限

//首先将基类的公有成员和保护成员当作在类定义中的普通成员，再加以限制符即可

/\*

has---a（基类由包含的类派生而来/类中含有类：相对于基类，范围变小）: 私有继承，基类的公有成员和保护成员将成为派生类的私有成员。

基类方法将不会成为派生对象公有接口的一部分，可以在派生类的成员函数中访问。

1、定义与初始化基类组件

class Student:private std::string,private std::valarray<double>

{

Student(const std::string &s, int n) :std::string(s), ArrayDb(n){};

……

}

2、访问基类（包含类）的方法（使用基类域限定符::）

double Student::Average()const

{

if (ArrayDb::size() > 0)

return ArrayDb::sum() / ArrayDb::size();

else

return 0;

}

return ArrayDb::operator[](i);//原型

3、访问基类对象（this指针+强类型转换）

const string & Student::Name()const

{

return (const string &)\*this;//基类对象本身为类对象(未知对象名称)，强制类型转换

}

4、访问基类的友元（转换为基类）

os<<（const String &）stu

5、使用using 重新定义访问权限（保护派生与有派生）

将函数调用包装在另一个函数调用之中：

class Student:private std::string,private std::valarray<double>

{

public:

using std::valarray<double>::max;

……

};

cout<<"high score: "<<ada[i].max<<endl;

is----a(由类拓展类：相对基类，范围变大)：基类的公有方法将成为派生类的公有方法，且派生类可以直接访问基类的保护成员。

保护继承：基类的的公有成员和保护成员将成为派生类的保护成员

class name1:privete classname2

{ };

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*多重继承\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

虚基类：一个类从两个不同的类那里继承了两个同名的类成员，则需要在继承的类中声明虚基类

如果一个基类Worker,公有继承派生（public）出两个类：Waiter，Singer，则则两个类在生成对象时会有各自的Worker副本。

当Waiter，Singer公有继承派生出另一个类SingerWaiter,因为Waiter，Singer各继承了一个Worker组件,因此SingerWaiter有两个waiter组件，在使用基类指针时，会出现二义性

虚基类使得从多个类（基类相同）派生出的对象只继承一个基类对象。

新的构造函数规则：

在定义非虚基类的派生类的构造函数时，信息可以自动传递，如：

class A{public:A(int n=0):a(n){}}

class B:public A{B(int m=0;int n=0):A(n),b(m){}}

class C:public B{C(int q=0,int m=0,int n=0):B(m,n),C(q){}}

如果Worker是虚基类，则在调用Singer和Waiter的构造函数后，还需要显式调用Worker的构造函数(不会自动传递)，这在非虚基类是不允许的。

SingerWaiter(const Worker &wk,int p=0,int v=Singer::other):Worker(wk),Waiter(wk,p),Singer(wk,v){};

\*/

/\*

单继承中，派生方法可以调用基类的方法，在多重继承中，每个类的直接祖先都有一个基方法，在调用时会出现二义性。

可以使用域解析运算符指定方法，或定义新方法。

对于使用虚基类的派生类而言，同定义时一样，需要使用直接祖先的方法以及虚基的方法。

如果方法是保护的，则只能在继承层次的结构中使用它。

总之：在祖先相同时，使用MI（多重继承）必须引入虚基类，并修改构造函数初始化列表的规则。

1、虚基类和基类混合使用，虚基类继承一个对象，非虚基类继承多个对象

2、定义虚基类的派生类，在新派生类中，虚基类的派生成员名优于虚基类。

3、必要时对类的名称进行限制

\*/

## 类模板

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*类模版（兼容性错误）\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*(<>中定义类型的关键字有class,typename,可在一个模版中含有多种类型变量)

函数模版：

1、定义

template <typename Type>//template <typename T1,typename T2=int>(默认参数)

返回值类型 函数名(形参)//这里的Typename T,相当于# Typedefine Typename T

{

函数体;

}

2、具体化/实例化：在使用时在函数名后加<typename>即可

使用模版显式：返回值类型 函数名<typename>(实参);

:返回值类型 函数名<>(实参);

自动识别：返回值类型 函数名（实参）;

未知？推测类型：

void ft(T1 x, T2 y)

{

decltype(x + y) z = x + y;//已知x与y类型，z与(x+y)类型一致

}

类模版：

template <class Type>//Type为类成员中的某一种类型

class 类名

{

类成员定义;

}

使用：

类名<typename> 对象;

类方法定义：

template <class Type>//Type为类成员中的某一种类型

返回值类型 类名<Type>::方法名()

{

方法体;

}

即与不用模版相比，在定义前加：template <class Type>,将域限定符改为：类<Type>::，最后将要泛化的类型名用Type替换即可。

模板实例化：

1、隐式实例化：

ArrayTP<int,100> stuff;

ArrayTP<double,30> \*pt;//a pointter,no object need yet

pt=new Array<double,30>;//now an object is needed

2、显式实例化：

定义：

template<>class classname <specialized typename>{……}；

例：

template<>class SortedArray<const char\*>{……};

SortedArray <const char\*> data;

3、部分实例化：

template<>class classname <>{};//template<>声明的是没有被具体化的类型参数，classname<>声明的是涉及到的具体参数；

例：

//general template

template<class T1,class T2,class T3>class Trio{};

//specialization with T3 set to T2

template<class T1,class T2>class Trio<T1,T2,T2>{};

//specialization with T3 and T2 set to T1\*

template<class T1>class Trio<T1,\*T1,\*T1>{};

给定上述声明，编译器将作出如下选择：

Trio <int,sgort,chas\*>t1;//use general template

Trio <int,short>t2;//use Trio<T1,T2,T2>

Trio <char,char\*,char\*>t3;//use Trio<T1,\*T1,\*T1>

\*/

//使用指针时要注意，既要创建存储指针变量的空间，也要指出指针变量所指地址值的空间

/\*

char \*p;//仅创建了指针变量，没有创建用于保存字符串的空间；

char po[40];//分配了空间，同时指针变量为char\*保存在栈中

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*指针栈\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

栈保存的应该是一个指针数组，而不能只是一个指针

为什么要给类创建对象时使用new？

因为该对象可以用来存储不同数据，可以更新，当它用来存指针时，必须化内存用于存储指针所指的变量。

也就是说要独立给类中的指针型变量分配两部分内存，如果不用New则只有一部分，当外界变量发生改变或者被销毁时，指针所指变量会改变或无效。

故应该复制副本在对象中。

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*递归使用模版\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

递归使用：

ArrayTP<ArrayTP<int,5>,10> twodee;

等价=》int twodee[10][5];

将模板用作参数：

template<template<typename T>class Thing>

其中template<typename T>class是类型，Thing是参数

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*模板类和友元\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//使用有对象参数的模版类友元时，参数必须是可实例化的对象，若为类模版必须使用实例化后的类模版

/\*

模板友元：

非模板友元（无对象参数）：基于类模版的友元函数,未指定类模版的具体实例

类定义：

template <class T>

class HasFriend

{

public:

friend void counts();//所有模板类的友元

friend void report(HasFriend<T>&)//bound template friend,注意类模版<T>

……

}

函数定义：不需要说明模版

template <typename T>

void counts()

{

函数体；

}

void report(HasFriend<int>&hf)

{

函数体；

}

（需要声明）约束（bound）模板友元：友元类型取决于类被实例化的类型

声明：

template <typename T>void counts();

template <typename T>void report(T &);//参数不需要说明类

类定义：

template <class TT>

class HasFriend

{

public:

friend void counts<TT>();//指定类模版实例类型

friend void report<>(HasFrirend<TT> &);

……

}

函数定义：需要说明类模版,相当于指定参数类型（类实例）的一般函数模版

template <typename T>

void counts()

{

函数体；

}

template <typename T>

void report(T &hf)

{

函数体；

}

非约束模板友元：友元的所有具体化都是类的每一具体化的友元

类定义：

template <class TT>

class HasFriend

{

public:

template <typename A>friend void counts();//

template <typename C,typename D>

friend void report(C &,D &);

……

}

函数定义：

template <typename A>

friend void counts()

{

函数体；

}

template <typename C,typename D>

friend void report(C &,D &);

{

函数体；

}

\*/

//非模板类友元函数：不需要类外声明，函数不受类模版限制（参数可以为任意可直接创建对象的类引用），直接定义；

//约束类模版友元函数：需要类外声明，类中声明要指定函数实例化所采用的类的实例化类型friend void counts<TT>()，类的实例化类型决定函数实例化类型，定义是要声明模版；形参声明类型即可，定义需指明类对象

//非约束类模版友元函数：不需要类外声明，类中声明要加模版（不同对象的实例化）;函数具体化类型决定了类模版的实例化类型，定义需要声明模版。声明定义均只需要指定类型

/\*

template <typename C, typename D>

friend void report(C &, D &);

\*/

//模版别名：

/\*

template <class T>

using arrtype=atd::Array<T,12>;

也可以用做：

typedef std::Array<T,12> arrtype;

用作其他类型重命名：

typedef const char\* pc1;

using pc2=const char\*;

\*/

## 队列

//队列不能用动态数组模拟，因为数组删除一个元素后，其余元素的位置不会改变，需要依次向前移动

//而链表的节点存储该节点信息以及指向下一个节点的指针

/\*

struct Node

{

Item item;//data stored

struct Node \*next ;//point to next Node

}

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*构造函数成员初始化\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//与在类定义时的初始化等价

/\*Queue::Queue(int qs):qsize(qs)//qsize为const int ,将其初始化为qs

{

front=rear=NULL;

items=0;

}

或

Queue::Queue(int qs):qsize(qs),front(NULL),rear(NULL),items(0)

{}

模版：

Classy::Classy(int n,int m):mem1(n),mem2(m),mem(n\*m+2)

{

//……

}

等价于：

Class Classy

{

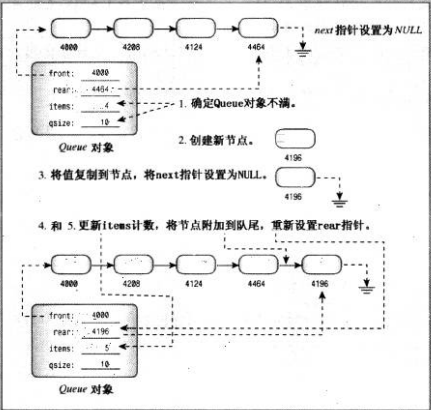
int mem1=n;

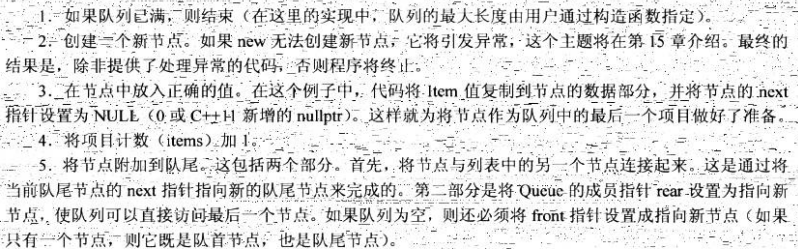
const int mem2=20;

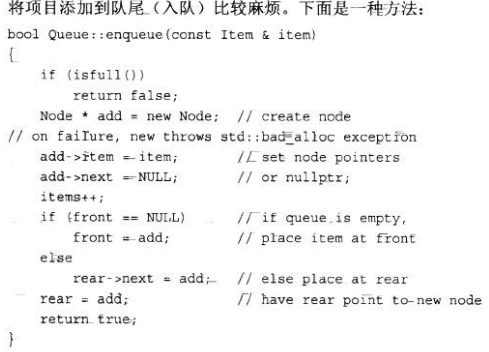
……

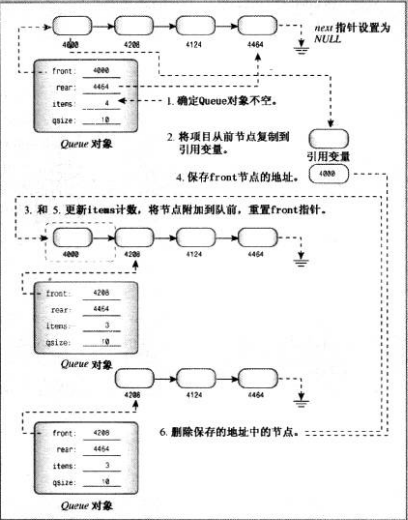
}

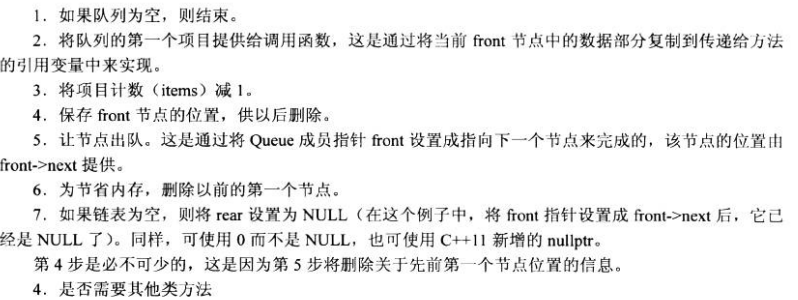
\*/

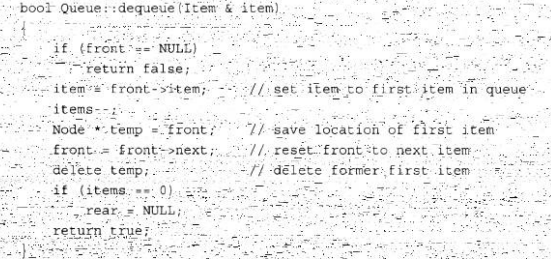












## 友元及异常

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*友元类\*\*\*\*\*\*\*\*\*88

/\*

在基类中声明为友元类即可

friend class Remote;

大多数时候只需要类中的某几个方法成为友元函数，而不需要全部函数

所以只需要在基类函数中声明友元函数同时考虑声明中含的类必须在编译器中

已经存在的原则，需要在基类定义之前对友元中含有的类进行声明，当该类中

含有基类成员时，还需要对基类进行声明。

class Tv;

class Remote{函数声明以及含类Tv的友元函数定义};

class Tv{类成员};

Remote的内联函数定义；

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*嵌套类\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

在另一个类中声明的类为嵌套类，它通过提供新的类型作用域来避免名称混乱；

包含类的成员函数可以创建和使用被嵌套的类的对象，而仅当声明位于公有部分

才能在包含类的外面使用嵌套类，而且必须使用作用域解析运算符。

嵌套类的访问权限：

1、作用域

在私有部分声明，只有公有方法才能访问派生类不能直接访问

在保护部分声明，在类中可以通过公有方法访问，派生类也可直接访问，但外部世界则不可见；

在公有部分声明，则允许类中，派生类和外部世界皆可见。

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*异常\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

0为分母时：

1、调用abort()异常终止

头文件<cstdlib>，会向标准错误流(cerr)发送错误消息，终止程序，返回错误值。

2、返回错误码

ostream类的get(void)成员通常返回下一个输入字符的ASCII码，但到达文件尾时，将返回特殊值EOF。

头文件<cfloat> 给 DBL\_MAX（最大浮点数）;

\*/

/\*

异常机制：

1、引发异常

程序异常，throw关键字表示引发异常，紧随其后的值指出了异常的特征

2、使用处理程序捕获异常

异常处理程序捕获异常，catch关键字表示捕获异常，随后使位于括号中的类型说明，它指出了异常处理程序

要响应的异常处理，然后是花括号括起来的代码块，指出要采取的措施。

3、使用try块

try块标识其中特定的异常可能被激活的代码块，它后面紧跟一个或多个catch块。

\*/

/\*

将对象用作异常类型：

有多个catch时，catch 与 throw后的值（对象或字符串）匹配

优点：可以使用不同的异常类型来区分不同函数在不同情况下引发的异常

首先执行try部分代码块，调用函数如果异常，按throw跳转路径返回try尾，

接着匹配catch 后的不同对象执行对应对象的方法，最后执行catch代码块部分。

注意：noexcept 关键字放在函数后面，表示该函数不会引发异常

函数fun()的返回语句将控制权返回到调用fun()的函数

throw语句将控制权向上返回到第一个这样的函数：包含能够捕捉相应异常的try-catch组合。

应发异常编译器将创建一个临时拷贝，即使异常规范和catch块中指定的是引用。

基类引用可以执行派生类对象。catch 基类引用,可以捕获throw 派生对象。

\*/

/\*

栈解退：

假设函数由于出现异常而终止，则程序将释放栈中的内存，但不会在释放栈的第一个返回地址后停止，

而是继续释放栈，直到找到一个位于try块中的返回地址。随后，控制权将转到块尾的异常处理程序，

而不是函数调用后面的第一条语句。该过程称为栈解退。

C++通常将信息放在栈中来处理函数调用。具体来说，程序将调用函数的指令的地址（返回地址）放在栈中，

当被调用的函数执行完毕后，程序将使用该地址来确定从哪里开始继续执行。另外，函数调用将函数参数放到栈中，

在栈中，这些参数被视为自动变量。如果被调用的函数创建了新的变量，则这些变量也被添加到栈中，如果被调用

的函数调用了另一函数，后者的信息将被添加在栈中。函数调用结束后，自动变量（对象会调用析构函数）将被释放，

同时栈顶元素也被释放，因此函数执行完后返回到调用它的指令或函数地址。

return返回时，返回到函数执行完下一条语句

throw 跳转到try的尾部。

\*/

/\*

生命周期：

对于全域对象，程序一开始，其构造式就先被执行（比程序进入点更早）；

程序即将结束前其析构式被执行。MFC 程序就有这样一个全域对象，

通常以application object 称呼。

对于区域对象，当对象诞生时，其构造式被执行；当程序流程将离开该对象的

存活范围（以至于对象将毁灭），其析构式被执行。

对于静态（static）对象，当对象诞生时其构造式被执行；当程序将结束时（此

对象因而将遭致毁灭）其析构式才被执行，但比全域对象的析构式早一步执

行。

对于以new 方式产生出来的区域对象，当对象诞生时其构造式被执行。析构

式则在对象被delete 时执行。

\*/

/\*

内存分配：

堆指：（满足堆性质的）优先队列的一种数据结构，第1个元素有最高的优先权；

栈指：满足先进后出的性质的数学或数据结构。

从静态存储区分配：此时的内存在程序编译的时候已经分配好，并且在程序的整个运行期间都存在。全局变量，static变量等在此存储

全局对象在编译时即分配内存并初始化，静态局部变量在程序第一次运行到它的表达式定义时才分配内存与初始化。

在栈区分配：相关代码执行时创建，执行结束时被自动释放。（自动变量）局部变量在此存储。栈内存分配运算内置于处理器的指令集中，效率高，但容量有限

在堆区分配：动态分配内存。用new/malloc时开辟，delete/free时释放。生存期由用户指定，灵活。但有内存泄露等问题

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*exception类异常\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

exception类：

头文件：

#include<exception>

含有虚方法what()返回一个字符串，可以在exception的派生类中定义。

class bad\_hmean :public std::exception{const char\* what(){return "错误信息1";}}

class bad\_gmean :public std::exception{const char\* what(){return "错误信息2";}}

如果想要以不同的方式处理这些派生而来的异常，可以在同一个基类处理程序捕获

try{}catch(std::exception & e){处理程序}

\*/

/\*

stdexcept异常类：

头文件：

# include <stdexcept>

定义了其他几个异常类，都是以公有方式从exception派生而来的。

典型的有logic\_error,和runtime\_error

1.logic\_error

class logic\_error : public exception{

public:

explicit logic\_error(const string & what\_arg);

……

}

class domain\_error : public logic\_error{

public:

explicit domain\_error(const string & what\_arg)

……

}

这两个类的构造函数接收一个string对象作为参数，该参数提供了方法what()返回的字符数据。

可以被用作两个派生类系列的基类，logic\_error描述了典型的逻辑错误

domain\_error:函数参数（定义域）的值不在定义域内。

invalid\_argument:函数传递了一个意料之外的错误，如传入应该是0或1，却传入除这两字符以外的其他参数

length\_error:没有足够的空间执行所需要的操作。

out\_of\_bounds:指示索引错误。

2.runtime\_error

该异常系列描述了可能在运行期间发生但难以预计和防范的错误。

underflow\_error:浮点数超过最小非零值

overflow\_error:整型和浮点型上溢出

range\_error:没有溢出，但是不在函数允许范围之内。

捕获：

try{}

catch(out\_of\_bounds &oe){……}//catch out\_of\_bounds error

catch(logic\_error &oe){……}//catch remaining logic\_error family

catch(exception &oe){……}//catch runtime\_error,exception objects

\*/

/\*

1、bad\_alloc异常与new:

头文件：

# include<new>

使用new导致的内存分配问题，继承的是exception。

catch(bad\_alloc &a){处理方法；cout<<a.what()<<endl;}

2、空指针与new

在new失败时可以选择返回空指针

int \* pi=new (std::nothrow)int [800];

if(pi==0){异常处理};

栈解退与内存分配：

void test(int n)

{

double \*ar=new double[n];

……

try{

if(oh\_no)

throw exception();

}

//没有这段代码的话，在栈解退时，new申请的内存不会被释放

// catch(exception &ex)

// {

// delete [] ar;

// throw;

// }

delete []ar;

return;

}

\*/

/\*

1.异常未被捕获：

在默认情况下，将导致程序异常终止。

异常未被捕获时，一般是调用terminate(),默认terminate()调用abort()函数。

可以使用set\_terminate()来设置；

# include <exception>

typedef void(\*terminate\_handler)();

terminate\_handler set\_terminate(terminate\_handler f) throw();//c++98

terminate\_handler set\_terminate(terminate\_handler f) noexcept();//c++11

void terminate();//c++98

void terminate()noexcept;//c++11

其中typedef 使terminate\_handler成为这样一种类型的名称：指向没有参数和返回值的函数的指针。

set\_terminate()函数将不带任何参数且返回值类型为void的函数的地址/名称做参数，并返回改函数地址

set\_terminate(myquit);//如果异常没被捕获将触发terminate(),随后调用myquit()

void myquit(){cout<<"terminate due to uncaught exception";exit(5);}

2.异常不符合异常规范

void unexpected();也会调用terminate()函数以及abort(),或自定义。

# include <exception>

typedef void(\*unexpected\_handler)();

unexpected\_handler set\_unexpected(unexpected\_handler f) throw();//c++98

unexpected\_handler set\_unexpected(unexpected\_handler f) noexcept();//c++11

void unexpected();//c++98

void unexpected()noexcept;//c++11

unexpected\_handler()比terminate\_handle()更严格

如果新引发的异常与异常规范不匹配，如果原异常规范没有bad\_exception类型，则调用terminate()

若有，则不匹配的异常被std::bad\_exception替代。

void myunexpection(){throw std::bad\_expection;//or just throw}

set\_unexpected(myunexpected);

try{}catch("1"){……}catch(bad\_expection &ex){……}

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RTTI(runtime type idenfication)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

RTTI：在程序运行期间确定对象的类型提供一种标准方式。

dynamic\_cast:将使用一个指向基类的指针生成一个指向派生类的指针，否则，该运算符返回0--空指针

typeid:返回一个指出对象类型的值

type\_info:结构存储了有关特定类型的信息。

1、dynamic\_cast:

可以回答“是否可以安全地把对象的地址赋给特定类型的指针”

Superb \* pm=dynamic<Superb \*>(pg)//pg能否转换为Superb对象类型

2、typeid:

typeid运算符返回一个对type\_info对象的引用。

3、type\_info：

重载了==与！=，以便对对象的类型(显示比较，不支持基类隐式转换)进行比较，一般都包含一个name成员，

返回一个随实现而异的字符串，通常（并不一定）是类的名称。

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*类型转换\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

为了防止一些松散的类型转换导致程序出错，使用以下规范：

dynamic\_cast:一般是类层次的向上转换

dynamic\_cast<type\_name>(expression);//type\_name:int class\_name,class\_name \*,char\*,……

const\_cast:用于执行只有一种用途的类型转换，即改变值为const/volatile，或删除const标签

const\_cast<type\_name>(expression);//type\_name与expression类型要相同

去掉const属性：const\_case<int\*> (&num)，常用，因为不能把一个const变量直接赋给一个非const变量，必须要转换。

变量本身的const属性是不能去除的，要想修改变量的值，一般是去除指针（或引用）的const属性，再进行间接修改。

加上const属性：const int\* k = const\_case<const int\*>(j)，一般很少用，因为可以把一个非const变量直接赋给一个const变量，比如：const int\* k = j;

static\_cast:支持向上转换，向下转换时会向另一方向转换，也是合法的。

枚举转换整型，double2int,float2long

reinterpret\_cast:天生危险的类型转换，不允许删除const

依赖于底层编程技术，是不可移植的，受数据长度和内存限制。

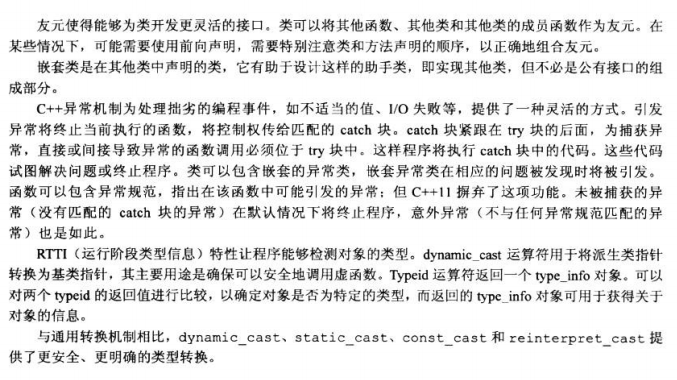
struct dat {short a;short b};

long value =0xA2234B76;

dat \* pd=reinterpret\_cast<dat \*>(&value);

cout<<hex<<pd->a;//display first 2 bytes of value.

\*/



## String类与标准模板

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*string类\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

1、构造字符串

7个构造函数，ctor标识

string(const char\*);

string(size\_type n,char c);//n个c

string(const string &str);//复制

string();

string(const char\*s,size\_type n);

string(const string &str,string size\_type pos=0,size\_typr=npos);//初始化为str中pos到尾，或pos开始的n个字符

string(string &&str)noexcept;//移动

string(initializer\_list<char>il)

template<class Iter>string(Iter begin,Iter end);

注意：string可以像字符数组一样访问字符，但是对象名不能像数组名一样做地址。

2、string输入

(1)C风格

cin>>cstr;

cin.getline(cstr,100);

cin.get(cstr,100);

(2)string对象,可以自动适应长度

cin>>str;

getline(cin,str)

注意getline都有一个可选参数，指定字符确定输入边界

getline(str,':');

cin.getline(cstr,100,':');

cin.opertor>>(fname);

operator>>(cin,fname);

在getline()限制string对象的长度：string::npos

(1)到达文件尾，输入的eofbit将被设置，fail()和eof()都将返回true.

(2)遇到分界字符(默认\n)，将分界字符从出入流中删除，但不存储。

(3)读取的字符数达到最大允许值（string::nops和可分配内存字节数中的较小值），设置输入流的failbit,fail()返回true.

string版本的operator>>()函数与此类似，只是它不断读取，直到遇到空白字符并将其留在队列中，直到遇到分界字符并将其丢弃。

空白字符指空格，回车，制表符，调用isspace()。

3、使用字符串

(1)重载比较运算符，从第一个字母开始按照机器排列序列（ASCII:数<大写<小写）

(2)确定字符串长度，返回字符串中的字符数：size(),length().

(3)搜索子字符或字符串：find()

size\_type find(const string &str,size\_type pos=0)const;//从字符串的pos位置开始，查找子字符串str，若有，返回首次出现索引，否则，返回string::npos.

size\_type find(const char \*s,size\_type pos=0)const;// 子字符串s，

size\_type find(const char \*s,size\_type pos=0,sizetype n);//查找s的前n个字符组成的字符串

size\_type find(char ch,type\_size pos=0)const;//查找字符ch

find()函数的拓展：

rfind():查找最后一次出现的位置

find\_first\_of():查找参数中任何一个字符首次出现的位置

find\_last\_of():查找参数中任何一个字符最后一次出现的位置

find\_first\_not\_of():查找参数中第一个不包含参数中的字符首次出现的位置

4、自动调整大小

在定义时会预分配一个内存大小（15），当字符串不断增大时，程序将分配一个大小

为原来两倍的新内存块，以提供足够大的空间。

capcity()返回当前分配给字符串的内存块的大小

reserve()方法能够请求内存块的最小长度。

如果想要用c风格字符串作为参数，而只有string对象时，可以调用c\_str()返回一个c风格字符串。

5、字符串种类

一般默认string基于char类型，事实上string库基于模版。

template <class charT,class traits=char \_traits<charT>,class Allocator=allocator<charT>>

basic\_string{……};

模版basic\_string有4个实例化，每个实例化具有一个typedef名称：

typedef basic\_string<char> string;

typedef basic\_string<wchar\_t> string;

typedef basic\_string<char16\_t> string;//c++11

typedef basic\_string<char32\_t> string;//c++11

traits类描述关于选定字符类型的特定情况，如如何对值比较。

Allocator类是一个管理内存分配的类。对于各种字符类型，都有预定义的allocator模版具体化，使用new和delete。

6、智能指针模版类

auto\_ptr,shared\_ptr,unique\_ptr

对于普通的指针变量，如果使用new申请 了内存，必须显式使用delete来释放，如果在定义异常时会跳过delete，也会导致

内存泄漏，因此考虑参考类的析构函数，希望在指针变量过期时，自动释放内存。

头文件：

# include<memory>

创建指针模版类实例，分配内存：

auto\_ptr<double> pd(new double);

所有智能指针类都有一个explicit构造函数，该构造函数将指针作为参数 (double \*p=new double;)

注意：不能在参数列表中使用&。

为什么要定义三种智能指针？

多个指针指向同一个内存，这是不允许的，因为变量过期时会释放多次内存

1、定义赋值运算符，深度复制，建立副本

2、建立所有权概念，只能有一个智能指针可以拥有它，让赋值操作转让所有权(operator=),转让的指针指向空地址会报错，auto\_ptr和unique\_ptr(更严格)

3、创建智能更高的指针，跟踪引用特定对象的智能指针，即引用计数。shared\_ptr

unique\_ptr不允许在程序中使用非临时智能指针变量的所有权转移（赋值），但运行临时智能指针变量的赋值（函数返回值为智能指针，模版实例化函数等）

这避免了非临时智能指针变量所有权转移之后的悬挂问题。

shared\_ptr和auto\_ptr使用new分配内存，不能使用new[]

unique\_ptr使用new和new[]

选择智能指针：

shared\_ptr：多个指针指向同一个对象。

unique\_ptr:不是多个指针指向一个对象，且函数使用new并返回指向该内存的指针

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*标准模板库（STL）:泛型编程\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

容器（同质）：存储值的类型相同

算法：完成特定任务(数组排序，链表查值)的处方

迭代器：用来遍历容器的对象，是广义指针。可以是指针，也可以是一个可执行类似指针操作的对象（解除引用，递增等等）

函数对象：类似于函数的对象，可以是类对象或函数指针。

\*/

/\*

模版类：vector

1、对应数组，区别于数组

定义： int array[n];

vector<int> a(n);

vector<int> b;

访问： array[i];a[i]

分配存储空间，自定义方法

2、头文件：

# include <vector>

template <class T,class Allocator=allocator<T>>

class vector{……};

3、可对矢量执行的操作

size():返回容器中元素的数目；

swap():交换两个容器的内容；

begin():返回指向容器中第一个元素的迭代器

end():返回一个表示超过容器尾的迭代器。

声明迭代器:

vector<double> ::iterator pd;

push\_back():将元素添加到矢量末尾；

erase():删除矢量中给定区间的元素，参数为迭代器；

insert():将另一容器指定区间元素插入指定区间

old\_v.insert(old\_v.begin(),new\_v.begin()+1,new\_v.end());

4、对矢量可执行的其他操作（STL函数）

for\_each(iterator1,iterator2,\*fun):将被指向的函数应用于容器区间中的各个元素，不可修改容器元素的值

random\_shuffle(iterator1,iterator2):随机排列两个区间的元素

sort(iterator1,iterator2，\*fun):排序，如果元素是用户自定义对象，需要重载<()

\*/

/\*

基于范围的for循环：

double prices[5]={4.499,10.399,45.872,45.345,78.987};

for(double x:prices)

cout<<x<<endl;

基于容器：

for (auto x : books)ShowReview1(x);

不同于for\_each(),基于范围的for循环可以修改容器的内容。

诀窍是指定一个引用参数：

void InflatReview(Review &r){r.rating++;}

for(auto &x:books)InflateReview(x);

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*泛型编程\*\*\*\*\*\*\*\*\*\*

//迭代器

/\*

模板使算法独立于存储的数据类型，而迭代器使算法独立于使用的容器类型。

vector<double>class:

vector<double>::iterator pr;

//声明vector<double>类的迭代器

1、迭代器类型

输入迭代器（只读）：输入是相对于程序的角度说的，来自容器的信息被视为输入：单向，可递增，读值

输出迭代器（只写）：将信息从程序传给容器

正向迭代器（读写）：支持++，多次通行算法，拥有输入输出迭代器所有特性。固定可重复排序

双向迭代器（读写）：支持前缀和后缀，拥有正向迭代器所有特性。a++,a--;

随机访问迭代器（读写）：支持随机操作，拥有双向迭代器所有特性。a[],a--,a+n;

2、迭代器层次结构

迭代器的类型是一种概念性描述

3、概念，改进，模型

概念可以具有类似继承的关系，但不是类的派生，是一种改进。

概念的具体实现被称为模型。

(1)指针是迭代器

sort(s.begin(),s.end());

sort(ARRY,ARRY+SIZE);

//使istream输入用作迭代器接口

//ostream\_iterator也有两个模版参数，第一个参数为要读取的数据类型，第二个参数指出输入流使用的字符类型。

第二个参数可以是C-Sty1e字符串，也可以是字符串常量。它用来表示各个元素被输出时的分隔符，默认情形下输出的各个元素并无任何分隔符

copy(dice.begin(),dice.end(),ostream\_iterator<int,char>(cout,""));

copy(istream\_iterator<int,char>(cin),istream\_iterator<int,char>(),dice.begin());

(2)其他有用的迭代器类型

reverse\_iterator:反向迭代器，执行递增操作会导致它被递减

vector类有一个名为rbegin()的成员函数和rend()的成员函数，前者返回指向超尾的反向迭代器，后者返回指向第一个元素的反向迭代器

rend()和begin()返回相同的值，但类型不同。

copy(c.rbegin(),c.rend(),out\_iter);

若rp指向6，则\*rp是5的位置

back\_insert\_iterator:用于允许在尾部快速插入的容器，vector满足

back\_insert\_iterator<vector<int>> back\_iter (dice);

front\_insert\_iterator:用于在起始位置做时间固定的插入的容器，queue满足，vector不满足

insert\_iterator：没有限制的任意位置插入元素

insert\_iterator<vector<int>> insert\_iter(dice,dice.begin());

\*/

//容器

/\*

1、种类

deque,list,queue,priority\_queue,stack,vector,map,multimap,set,multiset,bitset,

forward\_list,unordered\_map,unordered\_multimap,unordered\_set,unordered\_multiset

2、概念

容器是存储其他对象的对象。被存储的对象必须是同一种类型的，它们可以是OOP意义上的对象，也可以是内置数据类型

不能将任何类型的对象存储在在容器中，具体地说，类型必须是可以复制构造和可赋值的。

3、复杂度

从快到慢依次为：编译时间，固定时间，线性时间

编译时间：操作在编译时执行，执行时间为0；

固定复杂度意味着操作在发生运行阶段，但独立于对象中元素的数目

线性复杂度意味着时间与元素数目成正比。

4、序列

要求其元素按严格的线性顺序排列。

数组和链表都是系列，但分支结构不是。

5、类型

一般容器/非关联容器

(1)vector

头文件：<vector>

是数组的一种类表示，提供了自动内存管理功能，可以动态改变vector对象的长度

提供了对元素的随机访问，在尾部添加和删除元素的时间是固定的，但在头部或中间插入和删除元素的复杂度为线性时间

除序列外，vector还是可反转容器(reversible container)概念的模型，这增加了两个类方法：rbegin(),rend().

vector是最简单的序列类型。

(2)list

头文件：<list>

list模版类表示双向链表，可以双向遍历链表。

在链表中任意位置插入和删除的时间都是固定的。

除序列和可反转容器的函数外，链表还具有以下成员函数：

void merge(list<T,Alloc>,&x):将链表x与调用链表合并,线性时间

void remove(const T&val):删除val元素，线性时间

void sort()：排序,N个元素为NlogN

void splice(iterator pos,list<T ,Alloc>x):将x链表插入pos前面，固定时间,将原始区间移到目标地址，合并后，被合并列表为空，insert是插入副本

void unique():将连续相同的元素压缩为单个元素，线性时间。

remove\_if():访问区间所有元素，接收一个bool参数，如果为true则删除元素

(3)forward\_list

单链表，每个节点都只链接到下一个节点，而没有链接到前一个节点，因此只需要正向迭代器。

不同于vector和list，forward\_list是不可反转的容器。

(4)queue

头文件<queue>适配器类

不允许随机访问元素，不允许遍历元素，可以将元素添加到队尾，从队首删除，查看元素的值，可以在队首队尾检查元素数目以及测试队列是否为空。

方法：

bool empty()const;

size\_type size()const;返回元素数目

T & front();返回指向队首元素的引用

T & back();返回指向队尾元素的引用

void push(const T & x);在队尾插入x

void pop();删除队首元素

(5)priority\_queue

头文件<queue>适配器类

与queue不同的是，最大的元素被移到队首，内部区别在于，默认的底层类是vector.

可以修改用于确定哪个元素放到队首的方法，提供在一个可选的构造函数参数：

priority\_queue<int> pq2(greater<int>);//use greater<int>to order

(6)stack

头文件<stack>适配器类

不允许随机访问栈元素，不允许遍历栈。只允许栈顶访问，方法：

bool empty()const;

size\_type size()const;返回栈中元素数目

T & top();返回指向栈顶元素的引用

void push(const T & x);在栈顶插入x

void pop();删除栈顶元素

(7)array

头文件<array>并非STL容器

长度固定，无调整容器大小的操作。方法：

operator[]()，at()

STL方法：copy()for\_each().

关联容器：键值对

可以插入新元素，但不能指定位置，使用某种树来实现

(8)set

可反转，可排序，键是唯一的。

set<string> A;

set<string,less<string>> A;//第二个参数是可选的，可用来指示对键进行排序的比较函数或对象。

集合取并操作：

set\_union(A.begin(),A.end(),B.begin(),B.end(),ostream\_iterator<string,char>out(cout," "));

set\_union(A.begin(),A.end(),B.begin(),B.end(),insert\_iterator<set<string>>(C,C.begin()));

set\_intersection():交集

set\_differece():差集

lower\_bound():将键作为参数返回一个迭代器，该迭代器指向集合中第一个不小于键参数的成员。

upper\_bound():将键作为参数返回一个迭代器，该迭代器指向集合中第一个不大于键参数的成员。

(9)multimap

可反转，经过排序的容器，但是键和值的类型不同，同一个键可能与多个值关联

multimap<int,string> codes;//键的类型为int,值的类型为string

为了将信息结合在一起，实际的值类型将键类型和数据类型结合为一对，在STL中使用模版类

pair<class T,class U>将两种值存储到一个对象中。

pair<const int,string>item (213,"Los Angeles");

codes.insert(item);

或

codes.insert(pair<const int,string>item (213,"Los Angeles"));

获取multimap的信息(成员函数)：

count()接收键作为参数，返回具有该键的元素数目

lower\_bound(),upper\_bound()，确定上下限

equal\_range()用键作为参数，且返回两个迭代器，它们表示的区间与该键匹配，将两个迭代器封装在一个pair对象中。

pair<multimap<KeyType,string>::iterator,multimap<KeyType,string>::iterator> range=codes.equal\_range(718);

range.first,range.second可以访问两个迭代器；

无序关联容器：

unordered\_set,unordered\_multimap,unordered\_multiset,unodered\_map

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*函数对象(仿函数)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

函数符是可以以函数方式与( )结合使用的任意对象。这包括函数名，指向函数的指针，重载了( )的运算符的类对象

生成器是不用参数就可以调用的函数符

一元函数是用一个参数可以调用的函数符

二元函数是用两个参数就可以调用的函数符

返回bool值的一元函数是谓词

返回bool值的二元函数是二元谓词

预定义的函数符

支持将函数作为参数，以transform为例：

版本一：

transform(a.begin(),a.end(),out,sqrt);//计算容器中每个元素的平方根

版本二：

double add(double x,double y){return x+y;};

……

transform(a.begin(),a.end(),b.begin(),out,add);//a容器与b容器相加，需要为不同容器定义函数，所以创建模版

# include <functional>

plus<double>add;

double y=add(2.2,3.4);

自适应函数符和函数适配器：

函数符自适应的意义在于：函数适配器对象可以使用函数对象，并认为存在这些typedef成员

multiplies()函数可以执行乘法操作，但它是二元函数，因此需要一个函数适配器，将接受2个参数的函数符转换

为接受1个参数的函数符：

binder1st(f2,val)f1;//f1()：单参数,将常数赋给f1()的第一个参数

binder2nd(f2,val)f3;//将常数赋给f1()的第二个参数

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*算法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

对于算法函数设计，有两个主要的通用部分。

首先，用模版提供泛型，其次，使用迭代器访问容器中的数据通用表示。

1、算法组

头文件：<algorithm>

非修改式序列操作：不得修改容器内容，如:find(),for\_each()

修改式序列操作：可以修改容器内容和序列，transform(),random\_shuffle(),copy()

排序和相关操作:sort()和其他集合操作

头文件:<numeric>

通用数字运算：区间的内容操作，vector是最有可能使用这些操作的容器

2、算法的通用特征：使用迭代器和迭代器区间--------调用数组的函数

就地版本：结果被存放在原始数据的位置上，sort()

复制版本\_copy：接受一个额外的输出迭代器参数，将结果发送到另一个位置，copy()

\_if版本：将函数应用与容器元素得到的结果来执行操作

有些算法有两个版本：

template <class ForwadIterator,class T>

void replace(ForwardIterator first,InputIterator last,

const T&old\_value,const T&new\_value);

所有的old\_value替换为new\_value,就地发生的。

template <class InputIterator,class OutputIterator, class T >

OutputIterator replace\_copy(InputIterator first,InputIterator last

OutputIterator result,

const T&old\_value,const T& new\_value);

3、函数与容器方法

有时可以选用STL方法和STL函数，通常方法是更好的选择。首先，它适用于特定的容器，其次作为成员函数，它可以使用模版的

内存管理工具，从而在需要时调整容器的长度。

\*/

/\*

其他库：

vector模板类是一个容器类和算法系统的一部分，它支持面向容器的操作，如排序，插入排序，重新排序，搜索，转移数据等

valarray模板是面向数值计算的，为很多数学运算提供了一个简单、直观的接口

valarray<bool>vbool=number>9;//创建数组，vbool[i]=number[i]>9

valarray[slice(1,4,3)]=10;//slice对象作为索引表示一组，（开始索引，索引数目，跨距）

array为代替内置数组而设计的，让数组结构更紧凑，效率更高。

\*/

/\*

模板initializer\_list:(c++11)

头文件：<initializer\_list>

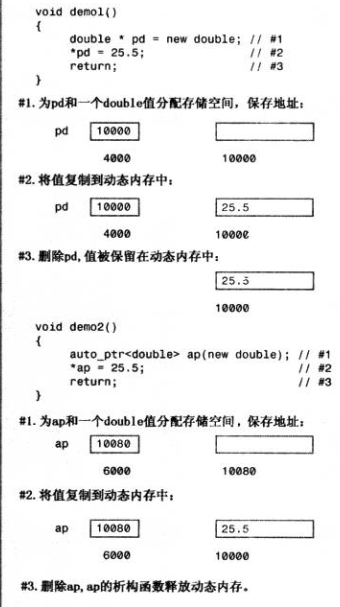
使用列表初始化语法将STL容器初始化为一系列值

std::vector<double> payments{45.99,56.78,67.8,89.01};

注意不能进行隐式的窄化转换

initializer\_list的迭代器类型为const

\*/









## 文件操作

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*IO流\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#pragma region iostream

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*标准输入流\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

使用 cin 从标准输入读取数据时，通常用到的方法有 cin>>、cin.get()，cin.getline()。

当我们从键盘输入字符串的时候需要敲一下回车键才能够将这个字符串送入到缓冲区中，

那么敲入的这个回车键（\r）会被转换为一个换行符（\n），这个换行符也会被存储在 cin

的缓冲区中并且被当成一个字符来计算！比如我们在键盘上敲下了 123456 这个字符串，

然后敲一下回车键（\r）将这个字符串送入了缓冲区中，那么此时缓冲区中的字节个数是 7 ，

而不是 6。

cin>>:

当 cin>> 从缓冲区中读取数据时，若缓冲区中第一个字符是空格、tab或换行这些分隔符时，

cin>> 会将其忽略并清除，继续读取下一个字符，若缓冲区为空，则继续等待。但是如果读取成功，

字符后面的分隔符是残留在缓冲区的，cin>> 不做处理。

不想略过空白字符，那就使用 noskipws 流控制。比如 cin>>noskipws>>input;

cin.get(char)

cin.get() 从输入缓冲区读取单个字符时不忽略分隔符，直接将其读取

cin.get() 的返回值是 int 类型，成功则返回读取字符的 ASCII 码值

cin.get(char\*s arsize )

遇到换行符时结束读取，但是不对换行符进行处理，换行符仍然残留在输入缓冲区。

将截取输入缓存区中的前n位

cin.get(char\* arsize delim)

指定读取的行数

cin.getline()

cin.getline 读取一行字符时，默认遇到 ‘\n’ 时终止，并且将 ‘\n’ 直接从输入缓冲区中删除掉，不会影响下面的输入处理。

//\*\*\*\*\*\*\*\*\*\*\*\*\*状态与清除\*\*\*\*\*\*\*\*\*\*\*

goodbit(0x0)：无错误

eofbit(0x1)：已到达文件尾

failbit(0x2)：非致命的输入/输出错误，可挽回

badbit(0x4)：致命的输入/输出错误，无法挽回

与这些条件状态对应的就是设置、读取和判断条件状态的流对象的成员函数：

s.eof()：若流 s 的 eofbit 置位，则返回 true

s.fail()：若流 s 的 failbit 置位，则返回 true

s.bad()：若流 s 的 badbit 置位，则返回 true

s.good()：若流 s 的 goodbit 置位，则返回 true

s.clear(flags)：清空当前状态, 然后把状态设置为 flags，返回 void

s.setstate(flags)：不清空当前状态，设置给定的状态 flags，返回 void

s.rdstate()：返回流 s 的当前条件状态，返回值类型为 ios\_base::iostate

getline()

getline() 利用 cin 可以从标准输入设备键盘读取一行，当遇到如下三种情况会结束读操作：

文件结束\遇到行分隔符/输入达到最大限度。

getline() 函数的参数使用了 string 字符串.

gets()

gets() 是 C 中的库函数，在头文件 <stdio.h> 申明，从标准输入设备读字符串，可以无限读取，不会判断上限，以回车或者文件结束符 EOF（ 即 -1） 结束

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*输出格式设置\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

cout.setf()的作用是通过设置格式标志来控制输出形式，

其中ios\_base::fixed表示：用正常的记数方法显示浮点数(与科学计数法相对应)；

ios\_base::floatfield表示小数点后保留6位小数。

setf()的第一原型：

C++为标准输入和输出定义了一些格式标志, 它可以通过flags(), setf(), 和 unsetf() 三个函数来控制.

例如：

cout.setf(ios\_base::left);//左对齐

cout << setiosflags(ios\_base::scientific);

第二原型:

包含两个参数，第一个参数和第一原型里的参数一样，第二个参数指出要清除第一参数中的哪些位，也就是说，

在第二原型中，第一个参数指出要设置哪些位，第二个参数指出要清除哪些位

cout.precision 跟 setprecision 一样，

cout.unsetf 跟 resetiosflags 一样。

setiosflags(ios::fixed) 固定的浮点显示

setiosflags(ios::scientific) 指数表示

setiosflags(ios::left) 左对齐

setiosflags(ios::right) 右对齐

setiosflags(ios::skipws 忽略前导空白

setiosflags(ios::uppercase) 16进制数大写输出

setiosflags(ios::lowercase) 16进制小写输出

setiosflags(ios::showpoint) 强制显示小数点

setiosflags(ios::showpos) 强制显示符号

boolalpha 可以使用单词”true”和”false”进行输入 / 输出的布尔值.

oct 用八进制格式显示数值.

dec 用十进制格式显示数值.

hex 用十六进制格式显示数值.

left 输出调整为左对齐.

right 输出调整为右对齐.

scientific 用科学记数法显示浮点数.

fixed 用正常的记数方法显示浮点数(与科学计数法相对应).

showbase 输出时显示所有数值的基数.

showpoint 显示小数点和额外的零，即使不需要.

showpos 在非负数值前面显示”＋（正号）”.

skipws 当从一个流进行读取时，跳过空白字符(spaces, tabs, newlines).

unitbuf 在每次插入以后，清空缓冲区.

internal 将填充字符回到符号和数值之间.

uppercase 以大写的形式显示科学记数法中的”e”和十六进制格式的”x”.

iostream 中定义的操作符：

操作符 描述 输入 输出

boolalpha 启用boolalpha标志 √ √

dec 启用dec标志 √ √

endl 输出换行标示，并清空缓冲区 √

ends 输出空字符 √

fixed 启用fixed标志 √

flush 清空流 √

hex 启用 hex 标志 √ √

internal 启用 internal 标志 √

left 启用 left 标志 √

noboolalpha 关闭boolalpha 标志 √ √

noshowbase 关闭showbase 标志 √

noshowpoint 关闭showpoint 标志 √

noshowpos 关闭showpos 标志 √

noskipws 关闭skipws 标志 √

nounitbuf 关闭unitbuf 标志 √

nouppercase 关闭uppercase 标志 √

oct 启用 oct 标志 √ √

right 启用 right 标志 √

scientific 启用 scientific 标志 √

showbase 启用 showbase 标志 √

showpoint 启用 showpoint 标志 √

showpos 启用 showpos 标志 √

skipws 启用 skipws 标志 √

unitbuf 启用 unitbuf 标志 √

uppercase 启用 uppercase 标志 √

ws 跳过所有前导空白字符 √

iomanip 中定义的操作符：

操作符 描述 输入 输出

resetiosflags(long f) 关闭被指定为f的标志 √ √

setbase(int base) 设置数值的基本数为base √

setfill(int ch) 设置填充字符为ch √

setiosflags(long f) 启用指定为f的标志 √ √

setprecision(int p) 设置数值的精度(四舍五入) √

setw(int w) 设置域宽度为w √

\*/

#pragma endregion

#pragma region ostream

/\*

C++把输入和输出看作字节流。输入时，程序从输入流中抽取字符；输出时，程序将字节插入到输出流中。

输入流的字节可能来自键盘，也可能来自存储设备（如硬盘）或其他程序。

同样，输出流的字节可以流向屏幕，打印机，存储设备或其他程序。流充当了程序和流源或流目标之间的桥梁

这使得C++程序可以以相同的方式对待来自键盘的输入和来自其他文件的输入。C++程序只是检查字节流，而不需要

知道字节来自何方。同样 ，通过使用流，C++程序处理输出的方式将独立于其去向。因此管理输入包含两步：

将流与输入去向的程序关联起来。

将流与文件关联起来。

通常，通过使用缓冲区可以更高效的处理输入和输出。缓冲区是用作中介的内存块，它是将信息从设备传输到

程序或从程序传输给设备的临时存储工具。C++通常在用户按下回车键后刷新缓冲区。

1、流，缓冲区和iostream文件

ios类（流属性，包括一个指向streambuf对象的指针）-----streambuf类（管理输入/输出缓冲区的内存）

派生:ostream类+istream类

派生类多重继承：iostream类

在程序包含iostream头文件时，自动创建8个流对象：

cin:标准输入流，被关联到标准输入设备（通常为键盘），wcin对象类似，但处理的是wchar\_t类型

cout:标准输出流，被关联到标准输出设备（通常为屏幕），wcout对象类似，但处理的是wchar\_t类型

cerr:标准错误流，可用于显示错误消息，默认被关联到标准输出设备。这个流没有被缓冲，wcerr对象类似，但处理的是wchar\_t类型

clog:标准错误流，默认关联标准输出设备，这个流被缓冲，wclog类型。。

对象代表流，该对象在头文件为程序声明一个cout对象时，将包含存储了与输入有关的信息的数据成员。

2、重定向

输入重定向<

输出重定向>

可以改变标准输出和标准输出，但是重定向不影响clog和cerr。

3、cout

(1)重载<<

与c一样，<<运算符的默认含义是按位左移

ostream & operator<<(int);//重载运算符接受一个int参数，函数返回一个ostream对象的引用。

默认可识别的类型：char,short,int,long,float,double..

c++用指向字符串存储位置的指针来表示字符串。

ostream类还为const signed char \*;const unsigned char\*;const char \*;void \*；定义了插入运算符函数。

(2)其他ostream方法

ostream & put(char);//显示单个字符

basic\_ostream<charT,traits>& write(const char\_type\* s,streamsize n);//显示整个字符串，指定输出长度

4、刷新缓存区

程序在接收到回车键消息之后会自动刷新缓存区

如果希望实现控制缓存区刷新，可以使用两个控制符中的一个来强行刷新：

flush：刷新缓存区

cout<<"Hello"<<flush;

flush(cout);

endl：刷新缓存区，并插入一个换行符

5、用cout进行格式化

(1)修改显示时使用的计数系统

[1]设置输出格式（ios\_base的成员函数）

ostream类由ios类派生而来，而ios类由ios\_base派生，ios\_base是包含char和wwxhar\_t具体化的模板

要控制输出数据以k进制显示，使用dec,hex,oct控制符:

hex(cout);

cout<<hex;

[2]调整字段宽度（成员函数）

int width();//返回字段宽度的当前设置

int width(int i);//将字段宽度设为i，并返回以前的字段宽度值

只影响接下来显示的一个项目，然后字段宽度恢复为默认值

[3]填充字符（成员函数）

默认情况下，cout用空格来填充字段中未被使用的部分，可以使用fill()成员函数来改变填充字符

cout.fill('\*');

[4]设置浮点数的显示精度（成员函数）

cout.precision(2);//默认指的是总位数，一直有效

[5]打印末尾的0和小数点(ios\_base成员函数)

ios\_base提供的setf()函数可以控制多种格式化特性

setf()函数的两个原型，均返回之前的格式设置

fmtflags setf(fmtflags);//fmtflags是格式常量，主要有：ios\_base::boolalpha,ios\_base::showbase,ios\_base::showpoint,ios\_base::uppercase,ios\_base::showpos

fmtflags setf(fmtflags,fmtflags);//第一个参数与上面一样，第二个参数指出要清除第一个参数中的哪些位

第二个参数 第一个参数 含义

ios\_base::dec

ios\_base::basefield ios\_base::oct

ios\_base::hex

ios\_base::floatfield ios\_base::fixed 定点计数法

ios\_base::scientific 科学计数法

ios\_base::left

ios\_base::adjustfield ios\_base::right 右对齐

ios\_base::internal 符号或基数左对齐，值右对齐

C++标准中，定点表示法和科学计数法都有下列特征：

精度指的是小数位数

显示末尾的0

ios\_base fmtflags old=cout.setf(ios::left,ios::adjustfield);

cout.setf(old,ios::adjustfield);//恢复以前设置

调用setf()的效果可以通过unsetf()消除，原型如下：

void unsetf(fmtflags mask);//其中，mask是位模式。mask所有的位都设为1，将使得对应的位被复位

cout.setf(0,ios\_base::floatfield);//默认模式

cout.unsetf(ios\_base::floatfield);//go to default mode,不管系统处于何种模式，都可以使用该语句回到默认模式

[6]标准控制符

使用setf()不是格式化对用户最为友好的方法，c++提供了多个控制符，可以调用setf()

cout<<left<<fixed;//打开左对齐和定点选项

其他控制符：

boolalpha,noboolalpha,showbase,showpoint,showpos,uppercase,internal,left,right,dec,oct,fixed,scientific

[7]头文件iomanip

setprecision():精度

setfill():填充字段,只针对下一个输出

setw():字段宽度

\*/

#pragma endregion

#pragma region istream

/\*

6、使用cin进行输入

cin>>value\_holder;

其中，value\_holder为存储输入的内存单元，它可以是变量，引用，被解除引用的指针，也可以是类或结构的成员

重载的>>运算符可以识别以下为类型：

signed char &,unsigned short &,unsigned int &,unsigned long long &,long double &

原型：istream & operator>>(int &)

可以将cin与hex,oct.dec控制符一起使用，来指定将整数输入解释为不同进制

istream类还为下列字符指针类型重载了>>抽取运算符：

signed char \*;

char \*;

unsigned char \*;

[1]cin>>如何检查输入

跳过空白（空格，换行符，制表符），直到遇到非空白符开始匹配，直到与目标不匹配的第一个字符之间的全部内容。

后面的输入将会留在输入流中。

[2]流状态

cin或cout对象包含一个描述流状态的数据成员（从ios\_base类继承）:

eofbit:当cin操作到达文件尾时，它将设置eofbit为1

failbit:当cin未能读到预期的字符时，它将设置failbit为1

badbit:在遇到一些无法诊断的失败破坏流时，badbit将被设置为1。

其他方法：

goodbit:另一种表示0的方法

good()：如果流看可以使用，则返回true

eof():如果eofbit()被设置，则返回ture

bad():badbit

fail():badbit或failbit

rdstate()：返回流状态

exceptions():返回一个掩码，指出哪些标记导致异常被引发

exceptions(iostate ex):设置哪些状态导致clear()引发异常；

clear(iostate s):将流状态设置为s，默认为0，如果（restate()&exceptions()）!=0,将引发异常

clear();//三位均清除

clear(eofbit);//设置eodbit,清除其他两位

setstate()：调用clear(rdstate()|s)，只影响参数中已设置的位

exceptions()方法会调用clear()方法，若返回值中某一状态位被设置，而当前状态中的对应位也被设置，则clear()将引发ios\_base::failure异常

ios\_base::failure异常类是由std::exception类派生的，因此包含一个what()方法。

如果输入了不恰当的字符，仅仅使用clear()清除位是行不通的，因为导致循环终止的不匹配输入仍在输入流中

程序必须跳过它。一种方法是使用isspace()函数是一个cctype函数，它在参数是空白字符时返回ture.

另一种方法是,丢弃行中的剩余部分，而不仅仅是下一单词

7、其他istream方法

（1）字符输入

get(char&)和get(void):提供不跳过空白的单字符输入功能，换行符仍保留在输入缓存区

istream & get(char&)；

int get(void)；

istream & get(char&)返回一个指向并调用它的istream对象的引用，可以拼接后面的其他抽取。文件尾返回false

cin.get(c1).get(c2).get(c3);

int get(void):使用返回值将输入传递给程序，到达文件尾返回EOF

如果程序需要跳过空白符，则使用>>方便，如果希望检查每个字符使用get()

（2）字符串输入：getline(),get(),ignore()

get(char\*,int,char)和getline()：读取整行（遇到换行符或读取字符数达到最大值），getline()自动删除换行符

istream & get(char\*,int,char);

istream & get(char\*,int);

istream & getline(char\*,int,char);

istream & getline(char\*,int);

第3个参数用于指定用作分解符的字符，默认为换行符。

istream & ignore(int,char);//(最大字符数，输入分界符)，读取并丢弃

原型：ignore(int=1,char=EOF)

注意：getline()会丢弃输入中的分界字符，get()不会

getline(char\*,int):如果没有读取任何字符（换行为一个字符），设置failbit

如果读取了最大数目字符，且行中还有字符，设置failbit

get(char\*,int):如果没有读取任何字符(空行也算)，则设置failbit

（3）其他istream方法

read():指定读取的字节，并存储在指定位置。不会在输入后加上空值字符，因此不能转换为字符串

istream & read(char\*,int);

peek():返回输入中的下一字符，但不抽取输入流中的字符，能够查看下一字符

ch=cin.peek()

gcount():返回最后一个非格式化（除>>外的抽取方法）抽取方法读取的字符数，

putback():将一个字符插入到输入字符串中，被插入的字符将是下一条输入语句读取的第一个字符

istream & putback(char)

\*/

#pragma endregion

#pragma region file

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*文件的输入与输出\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

头文件：# include <fstream>

1、简单的文件IO

要让程序写入（输出）文件，必须：

创建ofstream对象来管理输出流

ofstream fout;

将该对象与特定的文件关联起来

fout.open("jar.txt");//会自动创建文件

或者使用另一个构造函数直接将两步合为一步

ofstream fout("jar.txt");

以使用cout的方式使用该对象，唯一的区别是输出进入文件而不是屏幕

fout<<"Dull Date";

读取文件的要求：

创建一个ifstream对象管理输入流；

将该对象与特定的文件关联起来；

以使用cin的方式使用该对象；

//two statements

ifstream fin;

fin.open("jellyjar.txt");

//one statement

ifstream fis("jellyjar.txt");

char ch;

fin>>ch;

char buf[80];

fin.getline(buf,80);

当输入和输出流对象过期时，到文件的连接自动关闭，另外，也可以使用close()方法显示关闭。

fout.close();

fin.close();

检查流状态：继承了ios\_base类中的方法open(),如果试图打开一个不存在的文件进行输入，将设置failbit

与istream一样，在需要布尔量的地方ifstream会被转换为bool值。

2、打开多个文件

一般情况下，打开每个文件都会为其创建一个流；

批量处理时，可以打开一个流，将它依次关联到各个文件。

ifstream fin;

fin.open("fat.txt");

……

fin.close();

fin.clear();

fin.open("rat.txt");

……

fin.close();

3、命令行处理技术

wc report1 report2 report3

其中wc是程序名，report1,2,3是作为命令行参数传递给程序的文件名

c++有一种让在命令行环境中运行的程序能够访问命令行参数的机制：

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

argc为命令行中的参数个数，其中包括命令名本身，argv[]变量是一个指针数组，其中的指针指向命令行参数

argv[0]是一个指针，指向存储第一个命令行参数的字符串的第一个字符。

对于上述命令,则argc=4,argv[0]=wc,argv[1]=report1....

在VS中使用命令行的方法：项目--属性---调试，在这个对话框里，我们主要需要修改的就是命令参数和工作目录两项。

在命令参数中输入命令即可：$a.out paris.txt pair.txt

4、文件模式

文件模式描述的是文件将被如何使用：读，写，追加等，将流与文件关联时，都可以指定文件模式的第二个参数

ifstream fin("banjo",mode1);

ofstream fout();

fout.open("harp",mode2);

ios\_base类定义了一个openmode类型，用于表示模式；与fmtflags和iostate类型一样，它也是一种bitmask类型。

文件模式常量：

ios\_base::in r 打开文件，以便读取

ios\_base::out w 打开文件，以便写入

ios\_base::ate 打开文件，并移到文件尾

ios\_base::app append 追加到文件尾

ios\_base::trunc 如果文件存在，则短截文件，以前的内容将被删除

ios\_base::binary 二进制文件

(1)追加文件

对于文本格式与二进制的存储：（不仅结构体，同样也使用不适用虚方法的类）

结构体在以文本格式写入文件时，需要显式提供每个结构体成员，还必须将相邻的数据分隔开。

结构体在以二进制格式写入文件时，提供地址和结构体的长度，保存更为紧凑，精密。

使用二进制文件模式时，程序将数据从内存传给文件（反之亦然），将不会发生任何的隐式转换，而默认的文本文件并非如此。

使用read()和write()方法。

(2)随机存取

随机存取指的是直接移动（不是依次移动）到文件夹的任何位置。

fstream finout;

finout.open(file,ios\_base::in|ios\_base::out|ios\_base::binary)

fstream类为了满足文件中的移动方式，继承类方法：

seekg():将输入指针移到文件的指定位置

seekp():将输出指针移到文件的指定位置

basic\_istream<charT,traits>& seekg(off\_type,ios\_base::seekdir);//定位到离第二个参数指定的文件位置特定距离的位置

basic\_istream<charT,traits>& seekg(pos\_type);//定位到里文件开头特定距离的位置

事实上，由于fstream类使用缓冲区来存储数据，因此指针指向的是缓冲区的位置，而不是实际的文件。

也可以将seekp()用于ofstream对象，将seekg()用于ifstream对象。

类型升级：streampos和streamoff作为pos\_type和off\_Type的char的具体化，istream被模板basic\_istream取代

streamoff的值用来度量相对于文件特定位置的偏移量（字节）

seek\_dir参数是ios\_base类中定义的另一种整型，有3个可能的值：

ios\_base::beg指相对于文件开始处的偏移量

ios\_base::cur指的是相对于当前位置的偏移量

ios\_base::end指的是相对于文件尾的偏移量

如果要检查文件指针的当前位置，则对于输入流，可以使用tellg（）方法，对于输出流，可以使用tellp()方法。

(3)使用临时文件

头文件：# include <stdlib>

char\* tmpnam(char\* pszName);

(4)内核格式化

读取string对象中的格式化信息或者将格式化信息写入string对象中被称为内核格式化

头文件：# include <sstream>

该头文件定义了一个从ostream类派生的ostringstream类（还有一个用于宽字符集的类wostringstream）

可以将cout对象的方法用于ostringstream对象

格式化文本进入缓存区，该对象将使用动态内存分配来增大缓冲区。ostringstream类有一个名为str()的成员函数

该函数返回一个被初始化为缓存区内容的字符对象。

string mesg=outstr.str();

istringstream类允许使用istream方法族读取istringstream对象中的数据，istringstream可以使用string对象初始化

string facts;

istringstream instr(facts);

流类似于缓存区的作用，在需要时，可以从中抽取信息。

\*/

#pragma endregion

## C++新标准

/\*

C++11新特性：

1、新类型

新增了类型long long 和unsigned long long 以及支持64位的整型；

新增了类型char16\_t，char32\_t，以支持16位和32位的字符表示；

新增了“原始”字符串。

2、统一初始化功能

扩大了大括号的列表初始化范围，使用大括号时可添加也可不添加=

int x={5};

int ar\*=new int [4] {1,2,3,4};

class stump

{

public:

int root;

private:

stump(int r):root(r){}

};

stump s1{3};

stump s2={2};

(1)缩窄

使用初始化列表语法，编译器将禁止缩窄隐式转换

(2)std::initializer\_list

头文件：# include <initializer\_list>

c++11提供了模板类initializer\_list，可以将其用作构造函数的参数。

vector<int> a1 (10);//10个元素,并不是用10初始化容器；

vector<int> a2{12,23};//初始化

initializer\_list头文件提供了对initializer\_list模板类的支持。该类还包括成员函数begin()，end()

3、声明

(1)auto

自动类型推断：

auto mato=112;//int

auto pt=&mat0;//int \*

double fm(double,int);

auto pf=fm;//double(\*)(double,int)

简化模板声明若il是一个std::initializer\_list<double>对象

for(std::initializer<double>::iterator p=il.begin();p!=il.end();p++)

可以替换为：

for(auto p=il.begin();p!=il.end();p++)

(2)decltype

将变量的类型声明为表达式指定的类型：

double x;

int n;

decltype{x\*n} q;//q same as x\*n, double

定义模板：

template<typename T,typename U>

void ef(T t,U u)

{

。。。

decltype(T\*U) tu;

}

int j;

decltype(j);//int

decltype((j));//int &

(3)返回类型后置

double f1(double,int);

auto f2(double,int)->double;

模板作用域优化：

template<typename U,typename T>

auto eff(T t,U,u)->decltype (T\*U)

{

……

}

(4)模板别名：using

typedef std::vector<std::string>::iterator itType;

using itType=std::vector<std::string>::iterator;

using也可以用作模板部分具体化：

template<typename T>

using arr12=std::array<T,12>;

上述语句可以使array<T,int>（将参数int设置为12）.

std::array<double ,12>a1;

arr12<double>a1;

(5)nullptr(空指针)

(6)智能指针（自动释放内存）

头文件：# include <memory>

unique\_ptr;//所有权 new new[]

shared\_ptr;//多个指针指向同一对象，计数 new

weak\_ptr;

4、异常规范的修改

void f501(int)throw(bad\_dog);//can throw type bad\_dog exception

void f733(long long)throe()//doesn't throw an exception

新：

void f875(short,short)noexception;//doesn't throw an exception

5、作用域内枚举

(1)显式转换运算符

explicit :禁止自动类型转换

(2)类内成员初始化

可以使用大括号与赋值运算符

class session

{

public:

int mem1=10;

double mem2{1966.54};

short mem3;

private:

session(){};

session(short s):mem3(s){};

session(int n,double d,short s):mem1(n),mem2(d),mem3(s){};

……

}

6、模板和STL方面的修改

(1)基于范围的for循环

double prices[5]={1,2,3,4,5};

for(auto x:prices)

std::cout<<x<<std::endl;

(2)新增的STL容器

forward\_list，unordered\_map,unordered\_multimap,unordered\_set,unordered\_multiset

新增模板类array

std::array<int,34>ar;//array of 34 ints

(3)新增的STL方法

cbegin(),cend()

(4)valarray升级

面向数值计算的容器，添加了函数begin(),end()

(5)摒弃export

7、右值引用(实现移动语义)

传统的C++引用（左值引用）使得标识符关联到左值，左值是一个表述数据的表达式（变量或解除引用的指针），程序可以获取其地址

右值引用：&&

不能对其应用地址运算符的值。包括字面常量、诸如x+y的表达式以及返回值的函数（非引用）

8、移动语义和右值引用（修改了源对象，不能使用const）

移动语义的“移动”，意味着把某对象持有的资源或内容转移给另一个对象。实际文件还留在原来的地方，而只修改记录

要实现移动语义，需要采取某种方式，让编译器知道什么时候需要复制，什么时候不需要。这就是右值引用发挥作用的地方。

(1)构造函数：

移动构造函数（右值引用）：只记录

将源对象所有权（地址传递）给目标对象，并将其指针置空。

复制构造函数（const 左值引用）：深复制

(2)赋值

适用于构造函数的移动语义也适用于赋值运算符

删除目标对象的原始数据，将源对象的所有权转让给目标

(3)强制移动

法1：static\_cast<Useless &&>()

法2：

头文件：# include <utility>

std::move()

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*新的类功能\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

1、特殊的成员函数

在原有的4个特殊成员函数（默认构造函数，复制构造函数，复制赋值运算符和析构函数）的基础上，

新增了两个：移动构造函数，移动赋值运算符

移动构造函数原型：

someclass::someclass(const someclass &）；//copy

someclass::someclass(someclass &&）；//move

复制和移动运算符：

some & someclass::operator(const someclass &);//copy

some & someclass::operator(someclass &&);//move

2、默认的方法和禁用的方法

如果自己定义了相应的构造函数，则编译器不会自动创建默认构造函数，可以使用=defalt显式的声明默认版本

同样的，使用关键词=delete可以禁用方法

3、委托构造函数

定义构造函数时使用另一构造函数：

class1::class1(int nn):n(n);

class1::class1(double kk):class1(nn),k(kk){/do other stuff/}

4、继承构造函数

派生类可以继承基类构造函数的机制。

c++98:

namespace Box

{

int fn(int){...}

}

using Box::fn;

c++11:

class BS

{

private:

int q;

double w;

public:

BS():q(0),w(0){}

BS(int k):q(k),w(100){}

BS(double x):q(-1),w(x){}

BS(int k,double x):q(k),w(x){}

};

class QR:public BS

{

private:

short j;

public:

using BS:BS//QR类可以使用BS类的三个构造函数

QR():j(-100){}

QR(double x):BS(2\*x),j(int(x)){}

QR(int i):j(-2),BS(i,0.5\*i){}

};

int main()

{

QR o1;//QR()

QR o2(1.23);//QR(double)instead of BS(double)

QR (10,9.3);//not find in QR constructor,use BS(int,double)

}

5、管理虚方法：override和final

虚方法对实现多态类层次结构很重要，让基类引用或者指针能够根据指向的对象类型调用相应的方法

如果派生类的特征标不匹配，将隐藏而不是覆盖旧版本

override:放在参数列表后边，指出要覆盖一个虚函数，如果声明与基类不匹配将报错。

final:放在参数后面，禁止派生类覆盖特定的虚方法

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*lambda函数\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

1、比较函数指针、函数符（仿函数）、Lambda表达式|（函数对象）

函数符（仿函数）使用类来的重载来定义函数，使其可以适用多个对象。

lambda表达式---一种匿名函数

//lambda函数表达式（匿名函数）：auto 函数名=[capture](parameters)->return-type{body}

直接通过函数名调用

[捕获从句]（形参列表）-> 类型说明符｛函数体｝指明返回类型

[捕获从句]（形参列表)｛函数体｝未指明返回类型，自动识别

使用[]代替了函数名；

表达式完全由一条返回语句时，没有声明返回值类型，相当于使用decltype自动推算

否则需要声明返回值类型；

[](doubble x)->double{int y=x;return x-y;}//return double

例如：

bool f3(int x){return x%3==0;}

==>

[](int x){return x%3==0;}

可以给Lambda表达式指定名称：

auto mod3=[](int x){return x%3==0;}

bool result=mod3(100);

注意：函数指针阻止了内联，函数符和lambda表达式没有阻止内联

lambda表达式的其他功能：需要在定义函数时使用一些外部变量的捕获方式

[] // 沒有定义任何变量。使用未定义变量会引发错误。

[x, &y] // x以传值方式传入（默认），y以引用方式传入。

[&] // 任何被使用到的外部变量都隐式地以引用方式加以引用。

[=] // 任何被使用到的外部变量都隐式地以传值方式加以引用。

[&, x] // x显式地以传值方式加以引用。其余变量以引用方式加以引用。

[=, &z] // z显式地以引用方式加以引用。其余变量以传值方式加以引用。

[this]：通过引用捕获当前对象（其实是复制指针）；

[\*this]：通过传值方式捕获当前对象；

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*包装器\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

c++提供了多个包装器（wrapper/adapter）旨在提供更适合的接口

bind:可替代bind1st和bind2nd

men\_fn:将函数成员作为常规函数进行传递

reference\_wrapper:能够创建行为像引用但可被复制的对象

function:以统一方式处理多种类似于函数的形式

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*可变参数模板\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

接受可变数量的参数的模板函数或模板类

模板参数包

函数参数包

展开参数包

递归

1、模板和函数参数包（T t）

c++11提供了一个用省略号表示的元运算符，使用户可以声明表示模板参数包的标识符，模板参数包基本上是一个类型列表

同样，也可以声明函数参数包的标识符，而参数参数包基本上是一个值列表：

template <typename...Args>//Args is a template parameter pack

void show\_list1(Args...args)//args is a funtion parameter pack

{

...

}

2、展开参数包

template <typename...Args>//Args is a template parameter pack

void show\_list1(Args...args)//args is a funtion parameter pack

{

show\_list1(args...)

}

该函数调用与原始函数调用相同，会导致无限递归

3、在可变参数模板函数中使用递归

将函数参数包展开，对列表中的第一项进行处理，再将余下的内容传递给递归调用，以此类推，直到列表为空

template <typenameT,Typename...Args>//Args is a template parameter pack

void show\_list3(T value,Args...args)//args is a funtion parameter pack

每次调用参数数量减少1个

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*c++11新增的其他功能\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

1、并行编程

c++定义了一个支持线程化执行的内存模型，添加了关键字thread\_local

关键字thread\_local将变量声明为静态存储，其持续性与特定线程相关

库支持由原子操作（atomic operation）库和线程支持库组成。其中原子操作库提供头文件 atomic,

线程支持库提供了头文件thread,mutex,condition\_variable和future

(1)新增的库

random:随机数生成器，分布状态（均匀，二项，正态）

chrono:处理时间间隔

tuple:支持模板tuple，是广义的pari对象，pair对象可以存储两个类型不同的值，而tuple可以存储任意多个类型不同的值

ratio:支持有理数算术库

regex:支持正则表达式库

(2)低级编程

低级指的是抽象程度，意味着接近计算机硬件和机器语言使用的比特和字节

放松了POD(plain old data)的要求，允许共有体的成员有构造函数和析构函数

解决了内存对齐问题

(3)杂项

扩展整型：cstdint

自定义字面量机制：字面量运算符（literal operator）

提供调试工具assert(宏)

加强了对元编程的支持

2、语言变化

(1)Boost项目

(2)TR1(Tchnical Report 1)

包含Boost库的一部分，：模板类tuple,array,模板bind,function智能指针，static\_assert,regex库，random库

\*/