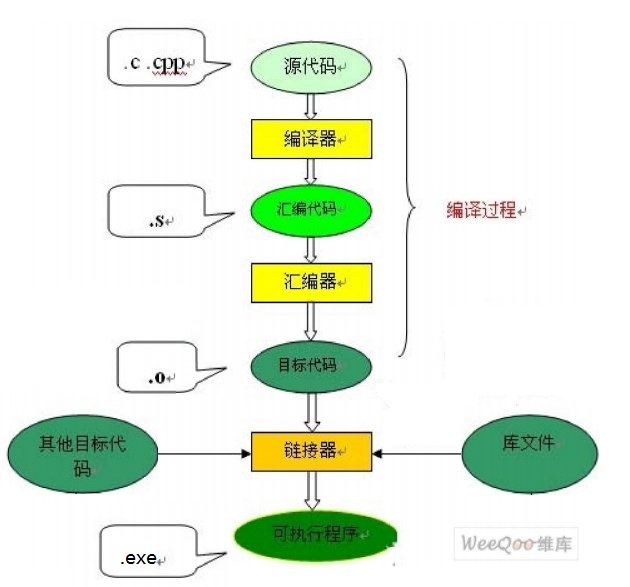
# 【编译】



【汇编】汇编语言仅高于机器语言，机器语言是01组成

【IDE的编译选项】

compile 编译当前文件

build和make 编译项目中修改过代码的文件

build all 重新编译项目中所有源代码

link 将编译后的源代码和库代码组合起来

run 运行程序

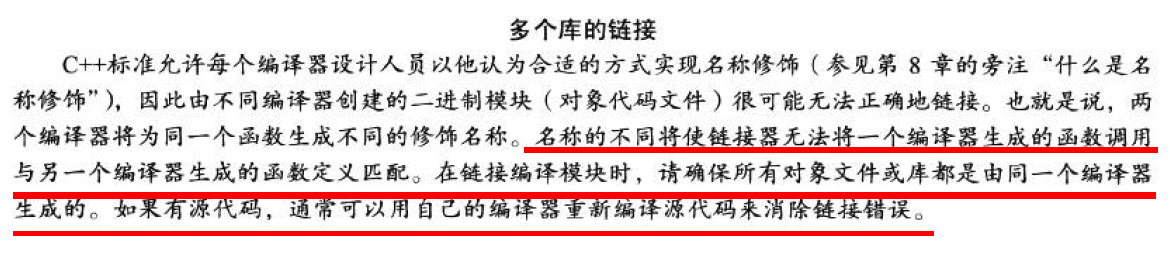
debug 以步进方式执行程序

调试版/发行版 包含额外代码，提供详细调试信息，但会降低执行速度

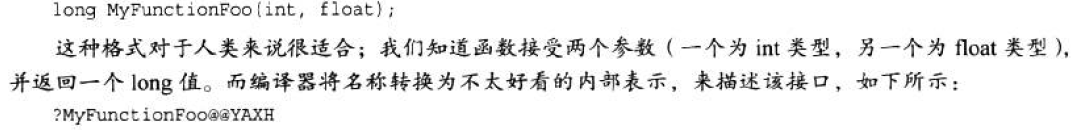
【#include】预处理器编译指令

【using namespace】编译指令

【链接】



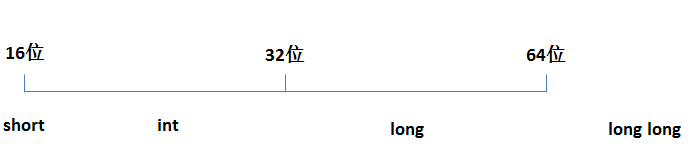
名称修饰



# 【简单类型】

## 【整形 integer】

|  |  |
| --- | --- |
| **short** | **16位** |
| **int** | **>= short** |
| **long** | **>= 32位，且>= int** |
| **long long** | **>= 64位，且>=long** |



【climits】

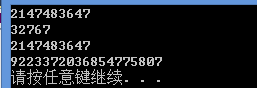
#include <climits> 定义了各整形的最大数值

cout << INT\_MAX << endl;

cout << SHRT\_MAX << endl;

cout << LONG\_MAX << endl;

cout << LLONG\_MAX << endl;



【超出最大值会发生什么？】

int a = INT\_MAX; //2147483647

a++;

cout << a << endl; //-2147483648

【sizeof】

cout << sizeof(int) << endl; //4

cout << sizeof(short) << endl; //2

cout << sizeof(long) << endl; //4

cout << sizeof(long long) << endl; //8

【初始化】各种奇怪的初始化

int a(1); //1

int b{ 2 }; //2

int c = { 3 }; //3

int d{}; //0

int e = {}; //0

C++11使大括号初始化器能够适用于所有类型。

【整形与进制】

cout << 42 << endl; //十进制 42

cout << 042 << endl; //八进制 34

cout << 0x42 << endl; //十六进制 66

## 【浮点型】

cout << sizeof(float) << endl; //4

cout << sizeof(double) << endl; //8

cout << sizeof(long double) << endl; //8

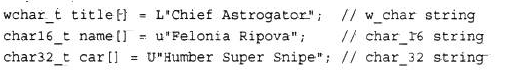
|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 有效位 | 指数位 | 符号位 |
| float | 0~22（23位） | 23~30（8位） | 31（1位） |
| double | 0~51（52位） | 52~62（11位） | 63（1位） |

【float的精度】6位有效数字（23位有效位，223=8 388 608）

【float的范围】约等于-3.4e38 ~ 3.4e38（8位指数位，-2128~2128）

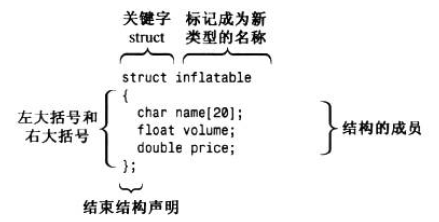
## 【字符型】char

char实际上是整形的一种，占8位。由于很多字符编码超过8位，所以C++还有wchar\_t类型，C++11增加了char16\_t和char32\_t类型，分别用L、u、U表示：



# 【复合类型】

## 【结构体】struct



//结构体类型

struct Book{

string name;

float price;

int page;

};

//结构体变量

struct {

string name;

float price;

int page;

}book01;

【初始化】大括号{}永远可行

Book book01{ "Little Prince", 35.5, 500 };

## 【共用体】union

结构体是“和”，共用体是“或”。

以下共用体可以存储int或double其中的一个类型

union Number{

int int\_value;

double double\_value;

};

Number num{11};

cout << num.int\_value << endl;//11

num.double\_value = 123.456;

cout << num.double\_value << endl;//123.456

共用体的内存大小是他最大成员的内存大小，上面的共用体内存大小和double一样。

共用体不能用string，共用体只能使用简单类型？

## 【枚举】enum

//定义枚举类型

enum Color{ red, green, blue, black, pink, orange };

//枚举常量，不打算创建变量

enum { red, green, blue, black, pink, orange };

## 【数组】

【初始化】{}大括号可以初始化任何类型（C++11）

int array01[4]{ 1, 2, 3, 4 };

int array02[4] = { 1, 2, 3, 4 };

int array03[4] = { 1, 2 };//1, 2, 0, 0

int totals[500] = { 0 };//全为0

【数组长度】

sizeof(array) / sizeof(int)

## 【字符串】

### 【字符数组 VS 字符串】'\0'

char str1[] = { '1', '2', '3', '4', '5', '6', '7'};//字符数组，长度为7

char str2[] = "1234567";//字符串，结尾含有'\0'，长度为8

用双引号括其的字符串隐式包含了结尾的空字符，因此不需要显式地包括它。

'S' 与 "S" 是不一样的： 'S' 是ASCII系统上83的另一种写法，而 "S" 是由两个字符组成的字符串。"S" **实际上是字符串所在内存地址（文字常量区）**，地址在C++中是一种独立的类型。

字符串常量"abcde"本质上是指向常量区的地址。当函数参数类型为const char \* s时，"abcde"是可以直接传递给形参的。

### 【字符串 VS 常量字符串】地址

char str1[] = "HelloWorld";//在栈区有分配空间

char str2[] = "HelloWorld";//在栈区有分配空间

//str1 != str2，因为两个地址不相同

//可以通过[]修改内容

char\* str3 = "HelloWorld";//常量字符串，在内存中只有一个地址（文字常量区）

char\* str4 = "HelloWorld";

//str3 == str4，因为两个指针指向相同的字符串常量

//不能通过[]修改内容,但可以修改指别的字符串

|  |  |
| --- | --- |
| char str[] | char\* str |
| sizeof 数组内存大小 | sizeof 指针变量内存大小 |
| strlen 字符数，不带空字符 | strlen 字符数，不带空字符 |
| [] 允许下标操作 | [] 不允许下标操作 |
| cout 字符串 | cout 字符串 |

对于字符串指针变量，cout自作聪明地替用户输出了字符串，而不是指针变量保存的地址。

### 【sizeof VS strlen VS cout】

char str1[] = "HelloWorld";

char str2[] = { 'H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd', '\0' };

char str3[] = { 'H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd' };

char\* str4 = "HelloWorld";

//sizeof根据它们的定义处理

cout << sizeof(str1) << endl;//11，数组

cout << sizeof(str2) << endl;//11，数组

cout << sizeof(str3) << endl;//10，数组

cout << sizeof(str4) << endl;//4，指针变量

//strlen把它们全部当做字符串处理

cout << strlen(str1) << endl;//10

cout << strlen(str2) << endl;//10

cout << strlen(str3) << endl;//缺少'\0'不可预知

cout << strlen(str4) << endl;//10

//cout把他们全部当做字符串处理

cout << str1 << endl;//HelloWorld

cout << str2 << endl;//HelloWorld

cout << str3 << endl;//缺少'\0'，一直向后输出

cout << str4 << endl;//HelloWorld

## 【string类】头文件<string> 命名空间std

### 【头文件】

STL中的许多头文件都间接地包含了<xstring>（但不要试图直接包含<xstring>），这就保证了仅include某些头文件（如#include <iostream>）就可使用std::string类。

然而，问题在于string类的很多操作，例如与cout相关的operator<<，是定义在<string>头文件中。因此必须手动地包含<string>头文件，才能使用它的全部功能。

### 【string类 VS char[]】

与使用数组相比，string更方便也更安全。

|  |  |  |
| --- | --- | --- |
|  | string | char [] |
| 长度 | str.size() | strlen(arr) |
| 复制 | str1 = str2 | strcpy(arr1, arr2) |
| 拼接 | str1 + str2 | strcat(arr1, arr2) |
| 比较 | str1 == str2 | strcmp(arr1, arr2) |
| 头文件 | <string> | <cstring> |

## 【字符操作】

【cctype】继承自C语言的，与字符相关的函数软件包

|  |  |
| --- | --- |
| isalpha(char) | 是否是字符 |
| isdigit(char) | 是否是数字 |
| ispunct(char) | 是否是标点 |
| isspace(char) | 是否是空格 |



【判断字母】



# 【指针】

计算机程序在存储数据时必须跟踪三种属性：

①信息存储在哪里 地址

②存储的值为多少 数值

③存储的信息是什么类型 类型

## 【指针变量】4个字节

int value = 1000;

int\* pointer = &value;

//整形变量value与指针变量pointer就像一个硬币的两面，通过\*和&翻到另一面

cout << ( &value == pointer ) << endl;

cout << ( \*pointer == value ) << endl;

【初始化的陷阱】

计算机分配给针变量用来储存地址的内存，但不会分配指针所指向的数据的内存。

为数据提供空间是一个独立的步骤，忽略这一步无疑是自找麻烦：

long\* fellow;

\*fellow = 2333;

fellow的确是一个指针，但是它指向哪里呢？这里并没有存储2333的内存，将会导致非常隐匿的错误。

## 【new】

【分配内存】allocation

//【申请】

long\* fellow = new long;

//【释放】

delete fellow;

fellow = nullptr; //【安全】

//【初始化】

int \*ptr = new int{ 6 };//指针指向的内存单元里面的数值为6

int \*ptr = new int(6);//同上

int \*ptr = new int[6];//注意这是数组，不是初始化

【注意】

①释放内存并不删除指针变量本身，它还可以指向其他内存

②new和delete要成对使用，否则将发生内存泄漏——分配的内存再也无法使用了

③不要尝试释放已经释放的内存，不过delete空指针是安全的

④不要创建两个指向同一块内存的指针（对于返回指针的函数，使用另一个指针）

【new为什么是动态申请内存？】

声明数组时，必先指明长度，因此数组的长度在编译时就已经决定好了。

为了在运行阶段决定数组长度，语言必须允许在程序运行时创建数组。C++采用的方法是使用new请求正确数量的内存，以及使用指针跟踪内存位置。

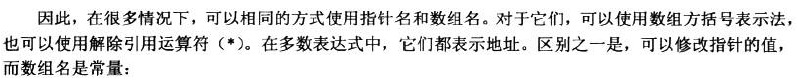
【动态数组】

int\* arr = new int[value]; //数组长度可以是变量，在运行时决定

delete [] arr;//注意方括号

arr = nullptr;

【数组指针与数组名】



C:\Users\knight\Desktop\360截图20180805223350070.jpg

【定位内存】<new>

new运算符除了【分配内存】的功能之外，还有【定位内存】的功能，能够指定要使用的内存位置。要使用定位功能，需要包含头文件<new>

#include <new>

static Type buffer[20];

Type\* ptr = new (buffer) Type;//把一个Type类型放到了buffer的内存当中？？

# 【文件操作】

【>>】输入操作符

数据从“外部”到“程序”

【<<】输出操作符

数据从“程序”到“外部”

## 【<iostream>】标准输入输出

|  |  |
| --- | --- |
| istream类 | cin |
| ostream类 | cout |

【格式化操作】

setf

precision

## 【<fstream>】文件输入输出

|  |  |
| --- | --- |
| ifstream类（继承istream类） | 从文件读取，可以使用cin的所有方法 |
| ofstream类（继承ostream类） | 输出到文件，可以使用cout的所有方法 |

### 【读取文件】ifstream

#include <fstream>

ifstream fin;

fin.open("1.txt");//关联文件

if (!fin.is\_open())//检查文件是否存在

{

cout << "could not open the file" << endl;

system("pause");

exit(EXIT\_FAILURE);

}

//……操作

fin.close();//关闭

#### 【逐字符读取】

char ch1, ch2, ch3;

fin >> ch1;//读取第一个字符

fin >> ch2;//读取第二个字符

fin >> ch3;//读取第三个字符

#### 【逐行读取】

char content[80];

fin.getline(content, 80);//读取一行（这一行已经被读了3个字符，这里将读取剩下的）

#### 【逐词读取】空格、换行都视为读入结束

vector<int> num{};

int temp;

while (fin >> temp)// 判断是否成功读入（读到EOF、非int都属于false）

{

num.push\_back(temp);

}

for (int i = 0; i < num.size(); i++)

cout << num[i] << endl;

#### 【读取整个文件】

【方法一】

ostringstream tmp;//#include <sstream>

tmp << fin.rdbuf();

string text = tmp.str();

cout << text << endl;

【方法二】

string text,tmp;

while (getline(fin, tmp))

{

text += tmp;

text += '\n';

}

cout << text << endl;

### 【写入文件】ofstream

#include <fstream>

ofstream outFile;//创建

outFile.open("1.txt");//关联文件

double temp = 123.456789;

int value = 123;

outFile << temp << endl;//输出double在第一行

outFile << value << endl; //输出int在第二行

outFile.close();//关闭

【文本IO和文本文件】

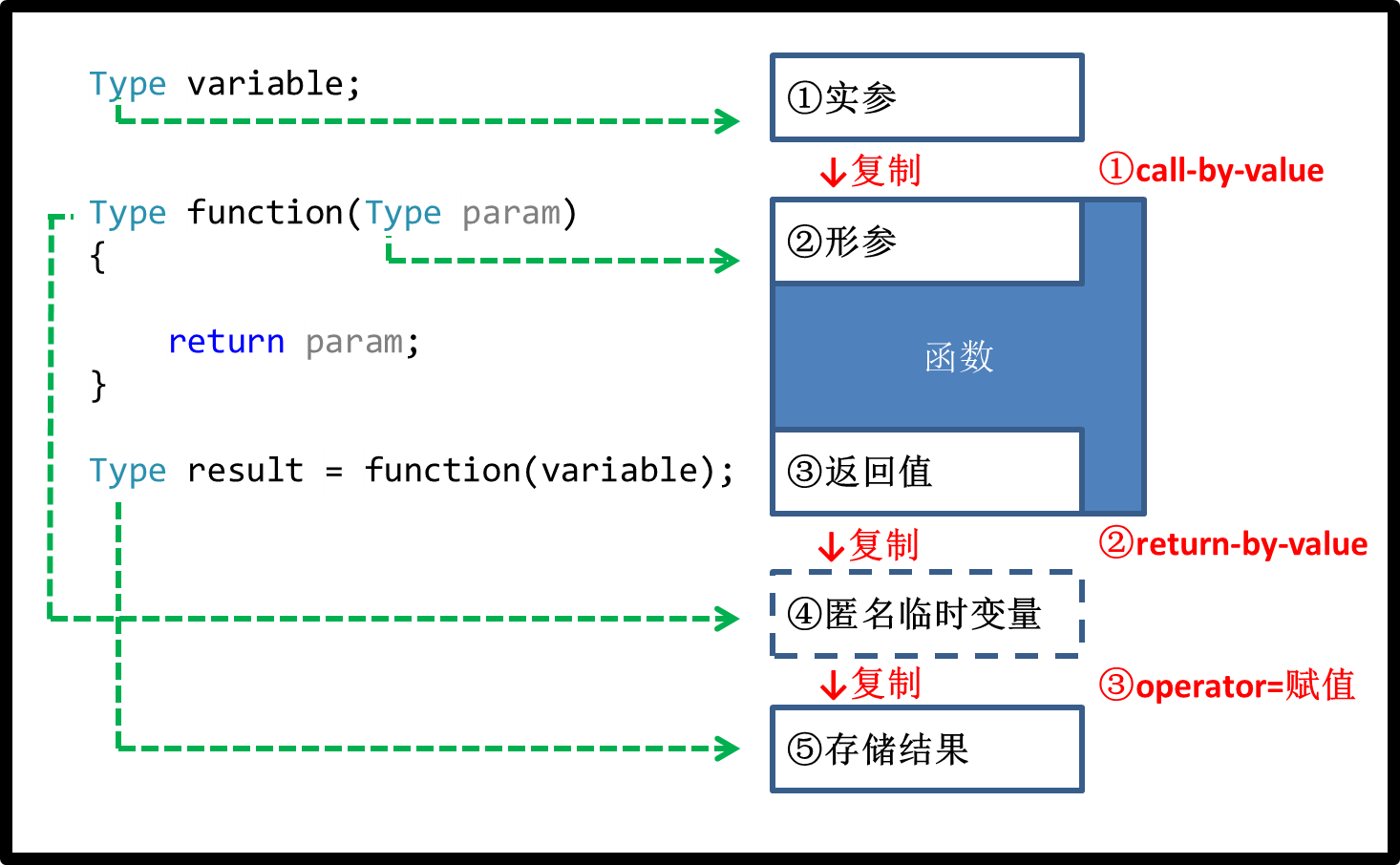
【cin】

cin输入时，程序将输入视为一系列的字节，其中每个字节都被解释为字符编码（二进制）。不管目标类型是什么，输入都是字符数据——文本数据，然后cin再负责将文本转成其他类型。

【cout】

# 【函数】

## 【3次复制、4个类型、5个变量】



## 【函数与数组】

### 【普通数组】

void function(int ptr[], int size)

//ptr被认为是指针变量，不能通过sizeof获得数组长度，所以数组长度要以另一个参数输入

void function(int \*ptr, int size)//等价

### 【二维数组】

int data[3][5] = { {1,2,3,4,5},

{4,5,6,7,8},

{7,8,9,0,1} };

int function1(int arr[][5], int size);//与下面等价

int function2(int (\*arr)[5], int size);//(\*arr)指向由“[5]个int”类型组成的数组

function1(arr, 3);

### 【字符串】

【字符串作参数】

void function(char\* ptr)

//如果输入的是指向字符串的指针变量，可以通过strlen或者判断'\0'确定长度

//除了判断 ch == '\0'之外，空字符还能当false用

char ptr[] ="123456789";

int i = 0;

while (ptr[i])//空字符还能当false用

{

cout << ptr[i] << endl;

i++;

}

【字符串作返回值】

不考虑string的话，函数无法返回字符串，只能返回字符串的地址，

## 【函数指针】void (\*ptr)(int);

与数据项一样，函数也有地址，函数的地址是存储其机器语言代码的内存的开始地址。可以编写将函数指针作为参数的函数，这样第二个函数可以找到第一个函数，并且调用它。

**声明函数指针的方法：返回值、参数与函数一致**

**获取函数地址的方法：函数名就是函数地址**

void function(int param);//函数声明

void (\*ptr)(int);//函数指针声明

ptr = function;//赋值（指针指向函数）

(\*ptr)(100);//调用（ptr是指针，那么\*ptr自然是指针指向的对象）

//使用auto

auto ptr = function;

//C++也允许直接通过指针名调用函数

ptr(100);

//不过这样做可读性并不高（ptr是指针而并不是函数名）

//但是也有其道理，因为function是存储函数的内存的首地址

//从ptr = function可以看出指针名与函数名等价

## 【内联函数】以“嵌入”代替“调用”

【概念介绍】

内联函数是C++为提高程序运行速度而做的一项改进。

首先了解代码运行的原理：

编译——产生可执行文件（里面全是二进制机器指令）

运行——将二进制机器指令加载到内存，每个函数有自己的内存位置。调用函数时，程序将跳转到另一个地址（函数地址），调用完函数之后再跳转回去。在内存中来回跳跃需要一定的开销。

内联——在编译时，将内联函数的二进制代码**直接嵌入**到调用的地方。好处是不需要跳转，速度快；坏处是占用更多的内存。**（如果程序在10个不同的地方调用同一个内联函数，则该程序将包含函数的10个二进制副本。）**

所以，内联适合**函数体短小**而**调用次数多**的函数。

【使用方法】

①加上关键字inline

②将整个定义写在声明的地方（如果函数体有多行，说明可能函数体太长，不适合内联）

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

【注意事项】

①程序员请求inline，但编译器不一定会实行，有可能编译器注意到函数太长

②递归的函数不能内联（编译时无法得知调用多少次，所以不知道要嵌入多少份）

【内联与宏】C++的内联功能远胜于C的宏定义

//内联函数

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

//宏定义函数

#define SQUARE(X) X\*X

内联函数使用上与函数一样，宏定义则只是简单的替换，如：

|  |  |
| --- | --- |
| SQUARE(12) | 正常 |
| SQUARE(v1 + v2) | **v1 + v2 \* v1 + v2 运算次序错误** |
| SQUARE(a++) | **a++ \* a++ 自增了两次** |

如果使用C语言的宏实现了类似函数的功能，应考虑将其转为C++。

## 【默认参数】

①格式：int function(int id, int age = 18, string name = "kira");

②默认值都在右边，当一个参数有默认值时，它的右边所有参数都要有默认值

错误示范：



③实参按从从左到右顺序赋给形参，不能跳过

错误示范：



④只需要在函数声明时设定默认值，函数定义时不需要

## 【函数重载】

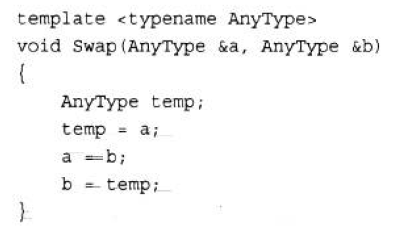
参数列表使得函数可以进行重载

参数列表也称作函数特征标，包括数量、类型、顺序

①强制类型转换时，当可以向多个类型转换，C++将因为不知向哪个类型转换而停止调用

②类型引用和类型本本身视为同一个特征标

## 【函数模板】



# 【左值与右值】

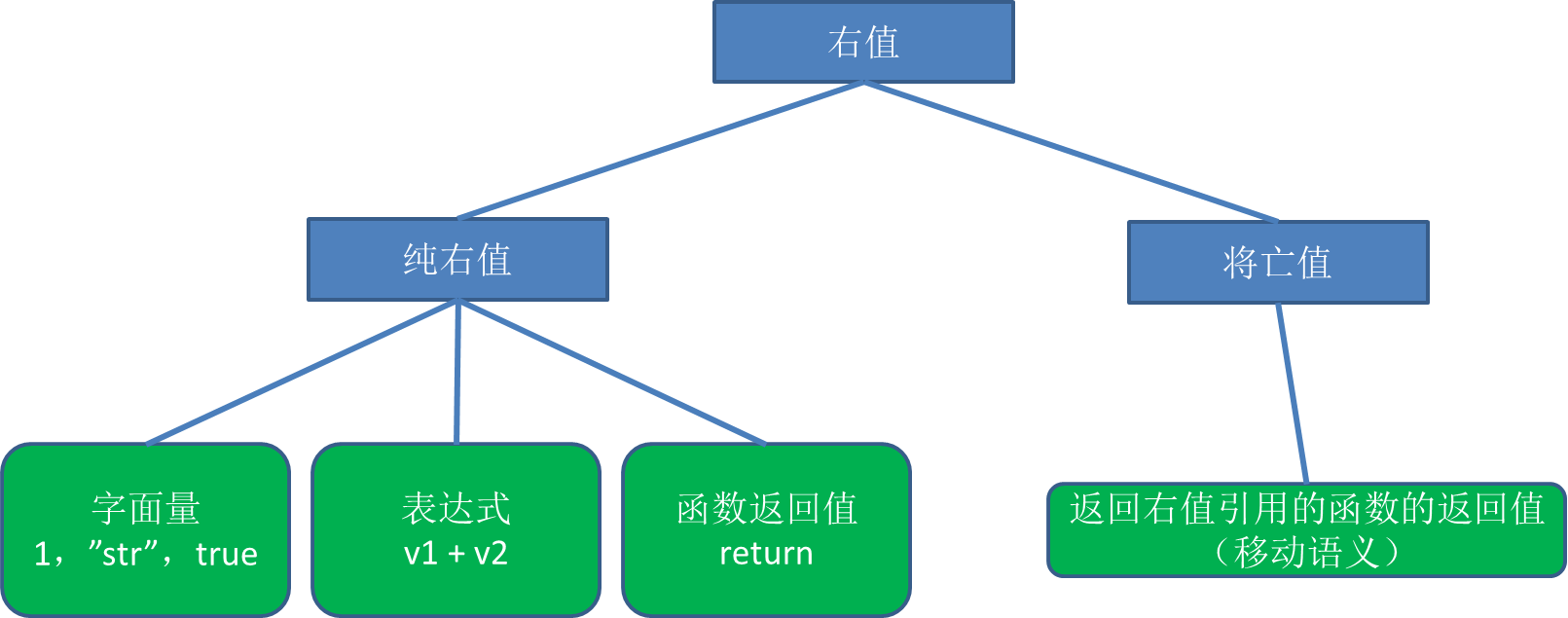
【区别】

①在等号左边的是左值；在等号右边的是右值。

②有名字，可以取地址的是左值；没有名字，不能取地址的是右值。

## 【左值】

## 【右值】



右值分为两种：（传统右值 和 移动的对象）

|  |  |
| --- | --- |
| 纯右值prvalue： | 例如： |
| 纯右值是传统意义上的右值，如上所述的在等号右边、没有名字、不能取地址 | ①字面量值（2，”str”，true） |
| ②运算表达式（a+b） |
| ③普通函数的返回值（非引用返回） |
| …… |

|  |  |
| --- | --- |
| 将亡值xvalue： | 例如： |
| C++11的新概念，将亡值是配合移动语义、右值引用而提出的概念  将亡值就是将要被移动的对象 | ①返回右值引用T&&的函数的返回值 |

编译器将返回值v和复制返回的匿名临时变量temp都识别为右值（纯右值？）

Type GetTemp()

{

Type v;

……

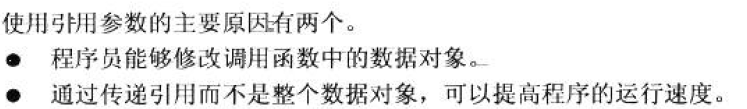
return v;//不明白，此处的v可以在等号左边、有名字、有地址，只是生命周期短

} //为什么会被认为是右值？

# 【引用】

【思想】

其实，引用主要是为了处理用户自定义类型（类、结构体），而不是基本的内置类型。



【引用 + 左右值 + const】组合

|  |  |
| --- | --- |
| int & | 左值引用 |
| const int & | **常量左值引用（万能引用）** |
| int && | **右值引用** |
| const int && | **常量右值引用** |

## 【左值引用——函数返回】

就像参数传入的时候，①要复制给②，当返回值传出的时候，③也要复制给④。

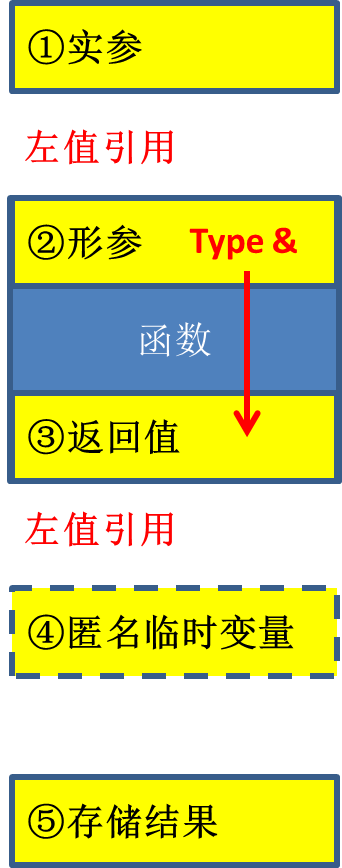
返回左值引用用于解决 ③返回值→④匿名临时变量 的复制问题。复制的效率低，用引用代替复制，效率更高。



但这样做**存在问题**，③在函数结束时就不存在了，但是④引用了③，就是说④引用了一个不存在的变量。

## 【左值引用——函数传参】

返回引用最重要的一点是，应避免返回函数终止时不再存在的单元的引用。最简单的解决方法是返回一个“作为参数传递给函数的引用”——引用传参，引用返回。



Type & function(Type & input)

{

......

return input;

}

【结论】return-by-left-reference要与call-by-left-reference

## 【常量左值引用——函数返回】

**解决返回左值引用的问题，防止对函数赋值**

【返回左值引用的问题】

Type & function(Type & input)

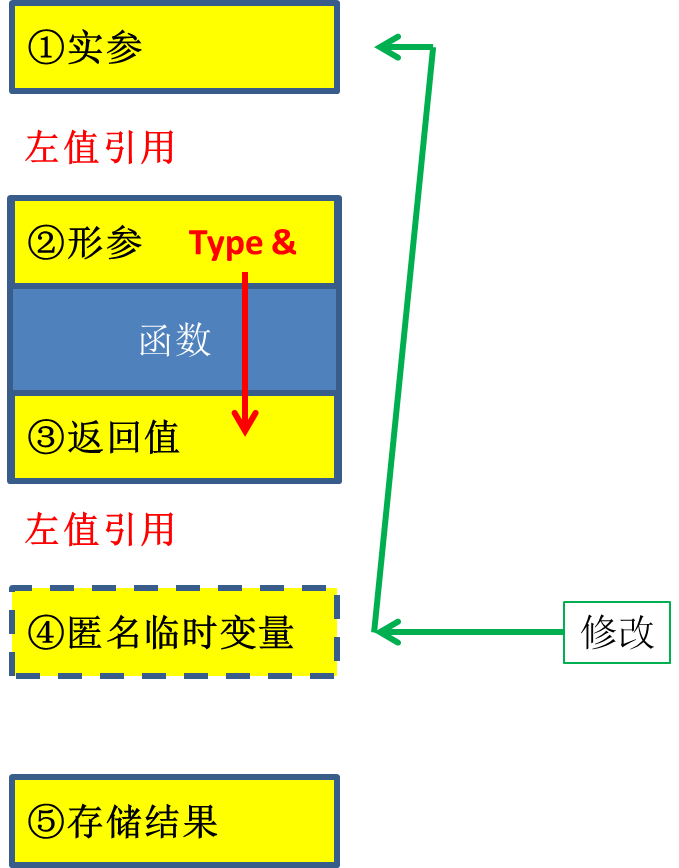
{

return input;

}

观察以上函数，当它被调用时，是可以作为左值被修改的：

function(v1) = v2;//函数返回值一般是右值，此时却是左值



const Type function(Type param);//加了const之后，不能被修改

## 【常量左值引用——函数传参】

//合并两个字符串的函数

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

{

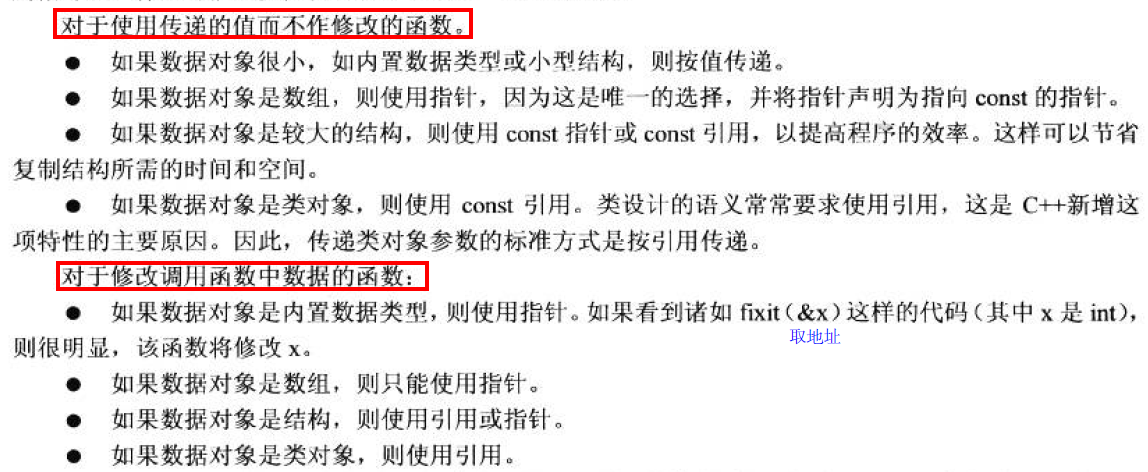
return s1 + s2;

}

对比函数string merge(string s1, string s2)，虽然也能完成任务，但是：

**引用传参效率更高，而const负责保护实参不被修改**

【传参时使用引用、指针、值传递的指导准则】



# 【各种语法符号】

## 【const】

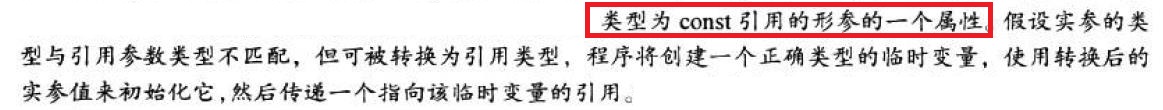
【尽可能使用const】

①使用const使函数能够处理const和非const的参数，否则只能处理非const数据

②const可以避免无意中修改数据

③const引用可以使函数能够使用临时变量

【形参为const引用为什么能处理非const的实参？】



【const与指针】

char ch = 'A';

const char\* ptr1 = &ch;//不能修改指针变量指向的值 \*ch是const

char\* const ptr2 = &ch;//不能修改指针变量的指向 ptr2是const

const char\* const ptr2 = &ch;//两个const

【const与成员函数】

当对象由const声明时，对象只能调用const方法：

声明：

void show()const;

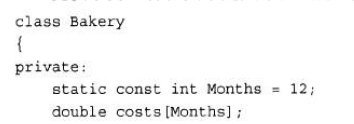
定义：

void Stock::show()const

与尽量使用const的原则一样，只要类方法不修改属性，就应该声明为const成员函数。

【const与对象属性】

对象属性具有类作用域。符号常量的作用域为类很有用，例如，类声明中可以用常量30表示数组长度，该常量对于所有对象都是一样的。



不能直接使用const int Months = 12，因为声明类时没有实例化对象，没有存储Months的空间，所以要声明为静态变量。（问题是为什么不直接使用12？）

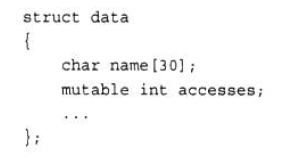
也可以通过枚举实现该功能：

enum{ Number = 10 };

## 【volatile】

const是不能修改，volatile则是即使程序代码没有修改内存单元，它的值也有可能发生变化。例如指针指向了某个硬件位置，硬件可能修改其中的内容。

## 【mutable】



在结构体或类中，可以用mutable指出，即使结构体被声明为const，其中的成员accesses也可以被修改。

## 【typedef】定义类型

【用法】

typedef unsigned long Item; //注意分号

后面使用到的声明用Item，以后想要修改unsigned long为其他类型时，只需要修改一处。相当于类型的宏定义。

【定义函数指针类型】

void function(int param);//函数声明

typedef void(\*Ptr\_fun)(int);//Ptr\_fun是一个类型

Ptr\_fun ptr = function; //ptr是指针

# 【switch】处理字符、数字类型、枚举

char num = 'B';

switch (num)

{

case 'A' : cout << "优";

break;

case 'B' : cout << "良";

break;

case 'C' : cout << "中";

break;

case 'd' ://实现多个条件

case 'D' : cout << "差";

break;

default:cout << "无法识别";

}

# 【程序的组织】

## 【头文件】

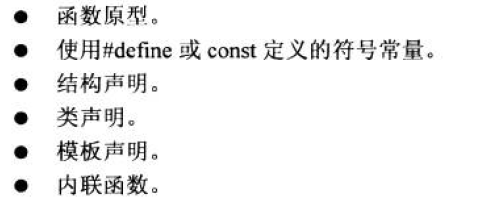
尖括号：查找系统存放标准头文件的位置

双引号：先查找当前工作目录，再查找系统存放标准头文件的位置

①不要将函数的定义或变量的声明放在头文件中。

如果头文件包含函数的定义，而有两个文件（同属于一个程序）包含了同一个头文件，那么一个程序中就包含了一个函数的两次定义。除非函数是内联的，否则将出错。

【头文件常包含的内容】



结构体不声明变量

模板声明不是将要被编译的代码

const数据和内联函数都有特殊的连接属性

### 【防止重复包含头文件】

①（新方式，可能有一些老的编译器不支持）

#pragma once

②

#ifndef \_HEADFILE\_H\_

#define \_HEADFILE\_H\_

#endif

### 【头文件扩展名】

.h的是标准C的头文件，没有.h的是标准C++的头文件，两种都是头文件。例如：

#include <iostream>//C++

#include <stdio.h>//C

新的C++标准库为了统一以前的C标准库，将原来C的标准库也放入了std名字空间，并把原来的C标准库去掉了.h后缀，前面加上字母“c”。例如 stdio.h就变成了cstdio，使用时为#include<cstdio>，其内容和#include<stdio.h>相同，但是要额外使用名字空间std。

### 【#include <cstdlib>】

cstdlib（即：stdlib.h）中常用到的函数：

srand函数：用于初始化随机数种子

rand函数：用于产生随机数

system函数：用于DOS系统功能调用

exit函数：用于退出程序

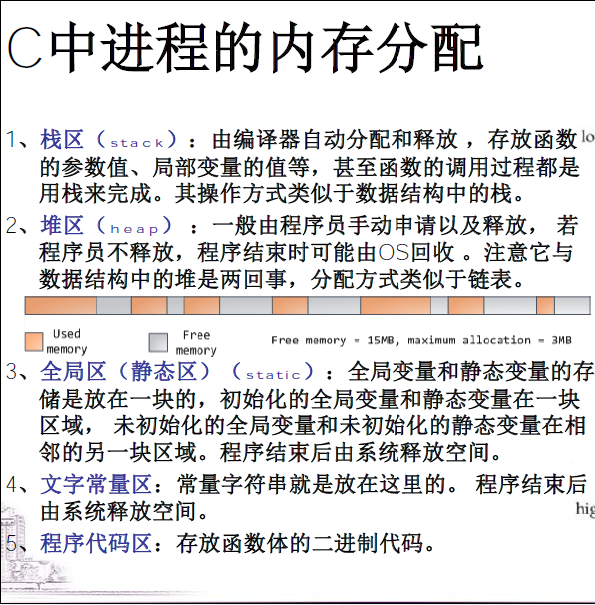
qsort函数：快速排序

itoa、atoi、atof等一系列转换函数（a是字符串）

malloc函数：（也可以用头文件malloc.h）动态分配内存

# 【内存模型】

## 【内存五区】



## 【储存时期】duration

|  |  |  |
| --- | --- | --- |
| 自动储存时期 | automatic | 代码块内定义{} |
| 静态储存时期 | static | 函数外部定义、static定义 |
| *动态储存时期* | dynamic | new |
| *线程储存时期* | C++11 | thread\_local |

**单独添加大括号括住自动变量，能决定它们的生命周期** 这个技巧用在对象上，能使对象在mian函数结束之前调用析构函数，显示一些关于析构函数的内容

## 【作用域】scope

|  |  |  |
| --- | --- | --- |
| 局部作用域（代码块） | block scope | 大括号{} |
| 全局作用域（文件） | file scope |  |
| *命名空间作用域* |  |  |
| *类作用域* |  |  |

## 【连接性】linkage

|  |  |  |
| --- | --- | --- |
| 无链接性 | no linkage | 在函数内部声明 |
| 外部链接性 | external linkage | 在函数外部声明（extern） |
| 内部链接性 | internal linkage | 用static在函数外部声明 |

【静态变量】

int global = 100; //①外部链接性，静态储存时期，全局作用域（文件）

static int file = 0; //②内部链接性，静态储存时期，全局作用域（文件）

void function()

{

static int block = 0; //③无链接性，静态储存时期，局部作用域（代码块）

}

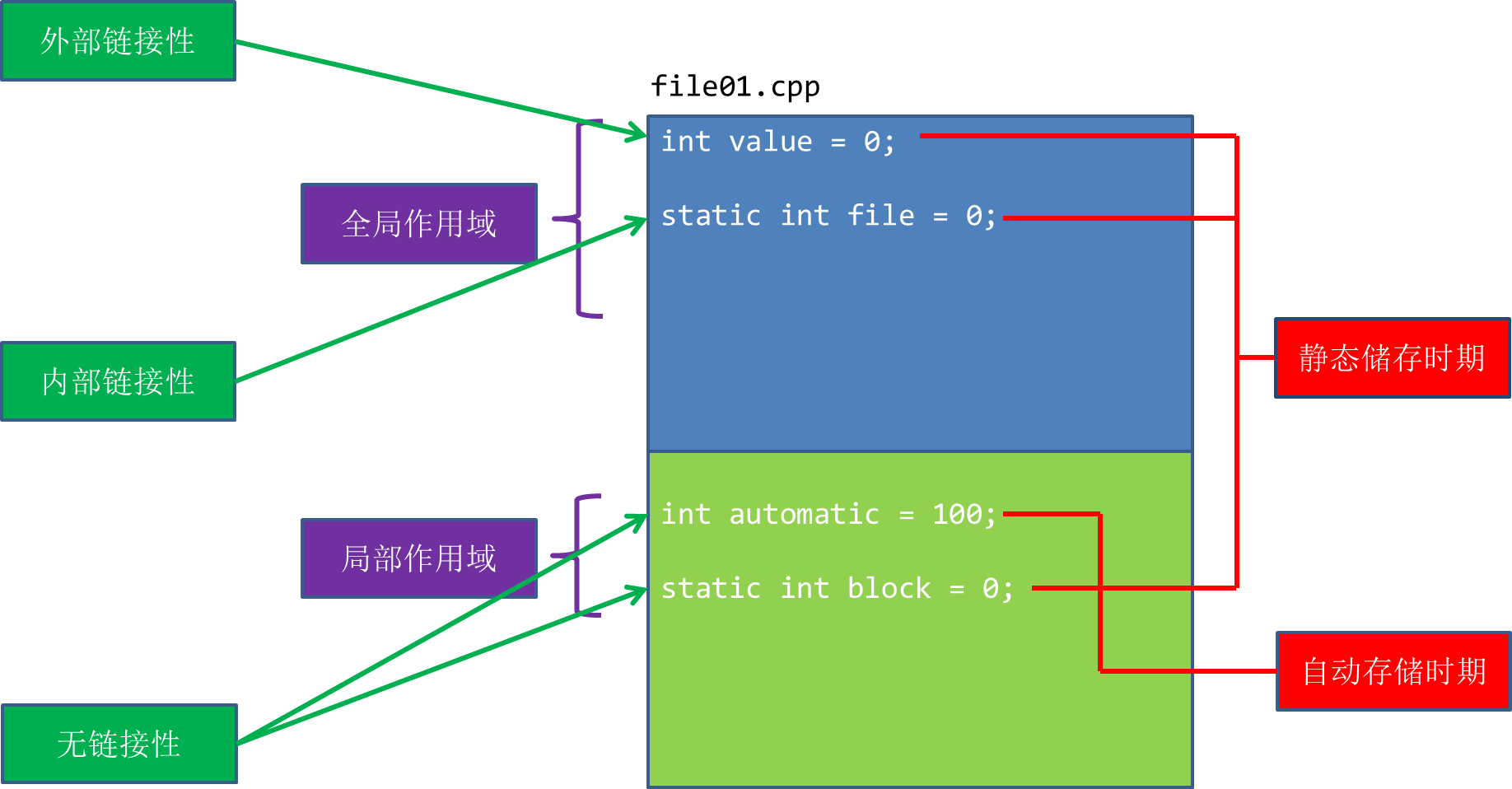
【注意】

static用在作用域为整个文件的声明时，表示内部链接性；用于局部声明时，表示储存时期为静态储存时期。

const全局变量是内部链接性，只能在文件中使用。让程序员更方便地将一组常量放在头文件中，并在同一个程序的多个文件使用，而不会产生重定义问题。

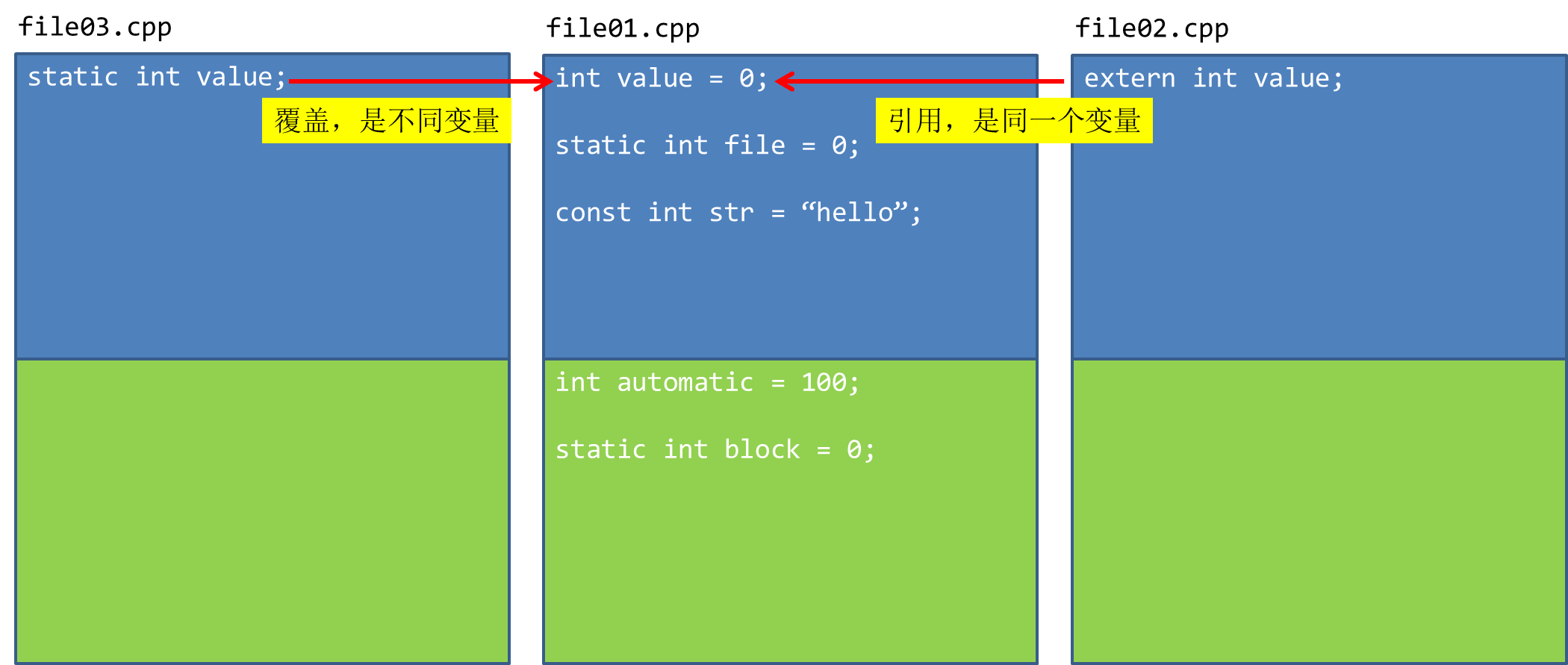
【总结】总结起来主要有四种情况（不讨论寄存器变量register）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 自动变量 | 自动储存时期 | 局部作用域 | 无链接性 | 局部变量 |
| 静态变量 | 静态储存时期 | 局部作用域 | 无链接性 |
| 静态储存时期 | 全局作用域 | 内部链接性 | 全局变量 |
| 静态储存时期 | 全局作用域 | 外部链接性 |



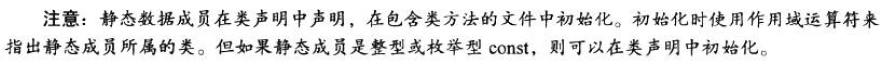
文件

函数



【其他情况】

①类中的static属性



在.h的类中定义，被所有实例化的对象所共享

在.cpp文件中初始化：初始化有int，没有static

int StringBad::m\_num\_strings = 0;//静态成员的初始化

不能在.h中初始化：

C:\Users\knight\Desktop\360截图20180823230032036.jpg在类中初始化的属性必须是const

②类中的const static属性

在.h文件中就已经初始化

作用与枚举常量一样

# 【命名空间】

【创建命名空间】

namespace kira{

int variable; //变量

void function() //函数

{}

struct Type{ //结构体

double value2;

string value3;

};

}

【使用】域解析运算符::

kira::variable//每次使用都添加kira::

using kira::variable//using将名称添加到当前区域中

using namespace kira//使命名空间所有名称在当前区域都可用

【嵌套命名空间】

namespace school{

namespace kira{

int variable;

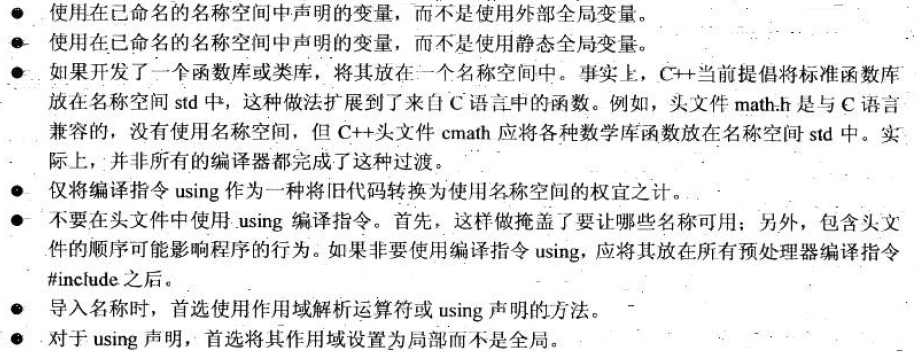
}

}

school::kira::variable;

using namespace school::kira;

【使用指导原则】

【面向对象】OOP

【运行时决策】

面向对象编程与传统的面向过程编程的区别在于，OOP强调在运行阶段（而不是编译阶段）进行决策。

声明数组时，必先指明长度，因此数组的长度在编译时就已经决定好了，这就是编译阶段决策。

new分配的数组可以

int\* arr = new int[value];

其中value是变量，在运行时才知道是多少，new允许动态分配内存，这将就是运行时决策。

【接口与实现分离】

接口在.h文件中，实现在.cpp文件中。

例如有一个矩形类Rectangle，它的成员函数get\_area()是获取属性m\_area的接口，接口只表示了返回面积的功能，而不管怎么实现。

可以是返回面积

return m\_area;

也可以是通过长乘以宽计算面积

return m\_length\*m\_width;

无论怎么修改实现方式，接口不变，就不用修改其它代码。

# 【类和对象】

C++对结构体进行了扩展，使之与类具有相同的特性。他们之间唯一区别就是，结构体的默认访问类型是public，而类的默认访问类型是private。

## 【构造函数】

【构造函数】初始化成员变量

Stock::Stock(const std::string & co, long n, double pr)

{

m\_company = co;

m\_shares = n;

m\_share\_val = pr;

}

### 【默认构造函数】——无参构造函数

定义对象时，没有提供参数，就会调用默认构造函数

①程序员没有定义构造函数时，编译器会自动生成【默认构造函数】，编译器提供的默认构造函数不做任何工作。

Stock::Stock(){ }//编译器提供的默认构造函数就像再这样，不做任何工作

②程序员定义了构造函数，就必须再提供默认构造函数（因为编译器不会再提供）

**【定义方法】**

**①重载一个无参的构造函数，在函数体为属性赋值**

Stock::Stock(const std::string & co, long n, double pr)//构造函数

{ …… }

Stock::Stock()//默认构造函数

{

m\_company = "no";

m\_shares = 0.0;

m\_share\_val = 0.0;

}

**②为已有的构造函数的参数提供默认值**

Stock(const std::string & co ="no", long n=0.0, double pr=0.0);

Stock(const std::string & co = "no", long n = 0.0, double pr = 0.0);

【构造函数与临时变量】



## 【析构函数】

Stock::~Stock(){}

在构造函数中new ，就要在析构函数中delete，nullptr

【调用】什么时候调用析构函数？？

通常应该由编译器决定什么时候调用。如果创建的对象是静态储存时期，则在程序结束时调用；如果创建的对象是自动储存时期，则代码块结束时调用；如果创建的对象是new产生的，则程序使用delete时调用。

## 【this指针】

【调用对象本身】

\*this

比如return \*this

【调用对象属性】

this->m\_member;

【调用对象方法】

this->function();

## 【类中的常量】

在.h文件中

enum{ weekdays = 5};//枚举常量

static const int weekdays = 5;//静态常量

## 【类中的静态变量】

类中的静态变量为所有对象所共有，在程序运行过程中，不论创建多少个对象，都只有一个静态变量，并且只初始化一次。

.h中声明：

static int m\_num\_strings;//实例的个数，无论多少个实例，共享一个变量

.cpp中初始化：

int MyString::m\_num\_strings = 0;//静态成员的初始化

## 【运算符重载】

如同\*作用于一个地址是取地址中的值，作用与两个数值是乘法。

编译器将根据操作数的数目和类型，决定使用哪种定义。

【示例】一个复数类

class Complex{

public:

Complex(double r = 0.0, double i = 0.0);

~Complex();

Complex operator+(const Complex & c)const;

private:

double m\_real;

double m\_image;

};

【**运算符表示法**与**函数表示法**】

Complex c3 = c1 + c2;

Complex c3 = c1.operator+(c2);

operator隐式调用c1（因为它调用了方法），显式调用c2（因为它作为参数传递）。

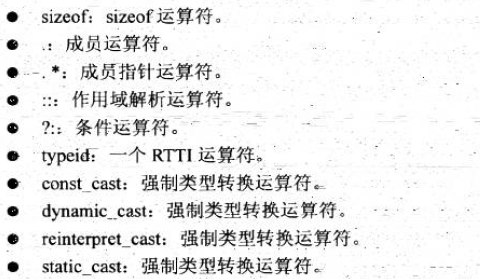
【限制】

①至少有一个操作数是用户自定义类型（防止为标准类型重载运算符）

②不改变运算符原有语法：操作数数量不变，优先级不变

③重载不是新建：只能重载已有的运算符，不能创造新的运算符

④不能重载以下运算符：



### 【重载加法运算符】operator+

Complex operator+(const Complex & c)const//适用于加减乘除

{

Complex result;

result.m\_real = m\_real + c.m\_real;

result.m\_image = m\_image + c.m\_image;

return result;

}

①重载为成员函数——隐式调用一个操作数，所以参数只有一个操作数

②不修改对象的属性——const函数

③返回值是局部变量，复制到匿名临时变量——返回Complex

【用构造函数处理返回新对象】

如果方法通过计算返回新对象（不一定局限于运算符重载），可以考虑用构造函数完成这个工作。这样一方面更加简洁，另一方面可以确保对象按照正确的形式创建。例如上面的例子可以改为：

Complex Complex::operator+(const Complex & c)const

{

return Complex(m\_real + c.m\_real, m\_image + c.m\_image);//相当于上面的result

}

【连续相加】允许

Complex c4 = c1 + c2 + c3;

Complex c4 = c1.operator+(c2.operator+(c3));

### 【重载数乘】operator\*

【成员函数的局限】

运算符重载时，第一个操作数是运算符的调用对象。面对数乘时，如果operator\*重载为成员函数：Complex operator\*(const double & num)const;将会发生以下问题：

c1 \* 0.5; //正确

0.5 \* c1; //错误

【友元函数】

这时候要**增加一个**友元函数，处理第二种情况（当然，两种情况可以用两个友元函数处理）：

class Complex{

public:

Complex(double r = 0.0, double i = 0.0);

~Complex();

//【成员函数】运算符重载

Complex operator\*(const double & num)const;//c1 \* 0.5

//【友元函数】运算符重载

friend Complex operator\*(const double & num, const Complex & c);//0.5 \* c1

private:

double m\_real;

double m\_image;

};

【注意】

①友元函数在类内声明（用friend），在类外定义（这时候不需要使用friend），它不是成员函数，所以声明时函数末尾不能加const，定义时也不能加::

Complex operator\*(const double & num, const Complex & c)

{

Complex result;

result.m\_real = c.m\_real \* num;

result.m\_image = c.m\_image \* num;

return result;

}

②虽然友元函数不是成员函数，但是具有与成员函数相同的访问权限。

Complex\*double与double\*Complex，从概念上看似完全相同的，但是前一个调用成员函数，后一个调用友元函数，这是C++语法的结果：语法规定，作成员函数时，类本身是第一个操作数，函数参数是剩下的操作数。

使用友元函数时，所有操作数都是由函数参数按顺序指定。

### 【重载输出运算符】operator<<

面对cout<<c1这样的需求，操作符两边是不同类型的操作数，一个办法是将operator<<作为ostream类的成员函数，因为第一个操作数（cout）是运算符的调用对象。但是修改iostream是一个危险的主意，因为会在标准接口上浪费时间。另一个方法是使用友元函数，考虑到c1不是<<的第一个操作数，把operator<< 作为Complex类的友元函数。看上去operator<<操作了ostream和Complex，需要是这两个类的友元函数，但是实际上operator<<不需要操作ostream的属性，只需要把cout当做一个整体使用，因此operator<<不需要ostream属性的访问权限，所以operator<<只需要是Complex的友元函数。

【实现】

类内声明友元函数：

friend ostream & operator<<(ostream & out, const Complex & c);

类外定义友元函数：

ostream & operator<<(ostream & out, const Complex & c)

{

out << c.m\_real << " " << c.m\_image;

return out;

}

①参数使用ostream & out，因为只能有一个ostream对象，就是cout。函数参数中的out是对cout的引用。

②返回ostream & 是为了实现连续输出：

cout << c1 << endl;

因为这相当于调用operator<<(cout, c1)然后返回cout的引用给<< endl;

而返回void只能实现：

cout << c1;

### 【重载负数运算符】operator-

减法运算符是二元运算符，负数运算符是一元运算符，重载可以为同一个运算符实现不同的行为。负数运算符重载实现如下：

【成员函数实现】

声明

Complex operator-()const;

定义

Complex Complex::operator-()const

{

return Complex(-m\_real, -m\_image);

}

【友元函数实现】

声明

friend Complex operator-(const Complex & c);

定义

Complex operator-(const Complex & c)

{

return Complex(-c.m\_real, -c.m\_image);

}

以上两种实现智能选其一，否则编译不通过，因为无法根据参数判断选择两种中的哪一个。

### 【总结】

**①成员还是友元**

由操作数类型确定。如果第一个操作数是类类型，可以选择重载为类的成员函数，后面的操作数作为成员函数参数。如果第一个操作数是标准类型，可以重载为友元函数。

选择为成员函数之后——

**②函数类型：是否修改类属性**

如果不修改类属性，如加+、减-、乘\*、除/，函数为const，使用传值返回

如果修改类属性，如自增++、自减—

③参数个数与类型：

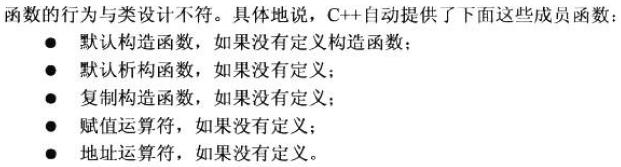
④返回值类型：能否满足连续运算，能否满足括号运算

## 【枚举模式】

Complex类的数据可以用两种形式表示，一种是实部和虚部，一种是复平面上的角度和半径。

## 【特殊成员函数①】

编译器自动生成的特殊成员函数：



地址运算符返回对象地址（this指针）

### 【编译器提供的 默认构造函数】

①编译器提供的默认构造函数不做任何事情

Complex::Complex();

{ }

②如果用户定义了构造函数，编译器将不再提供默认构造函数

### 【编译器提供的 析构函数】

只释放普通成员变量占用的空间，不能释放new申请的空间。

### 【编译器提供的 拷贝构造函数】

编译器提供的拷贝构造函数只实现浅拷贝（构造出新对象）

### 【编译器提供的 赋值运算符】

同样只实现浅拷贝（赋值给已有对象）

### 【编译器提供的 地址运算符】

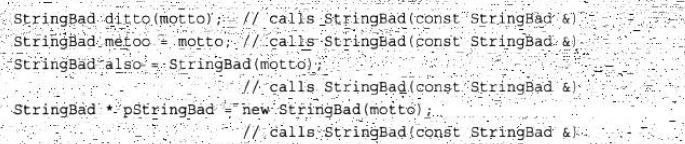
return this;

### 【自定义 拷贝构造函数】

需要知道两点，何时调用和有何功能

①何时调用（初始化的时候）

函数参数传值调用



（待确定）

②有何功能

逐个复制非静态成员变量。如果成员本身就是类对象，将再调用这个成员对象的拷贝构造函数。

【浅拷贝与深拷贝】

浅拷贝：只拷贝指针

深拷贝：拷贝指针指向的值

//拷贝构造

MyString::MyString(const MyString & st)

{

m\_len = st.m\_len;

m\_str = new char[m\_len + 1];

std::strcpy(m\_str, st.m\_str);

m\_num\_strings++;

std::cout << "构造第" << m\_num\_strings << "个对象" << std::endl;

}

### 【自定义 赋值运算符】

由于没有产生新对象，与拷贝构造函数有所不同

①delete对象之前分配的内存（？？）

②检测自己给自己赋值的情况（在赋值前内存可能已经delete）

③考虑到连续运算和括号运算的情况，返回引用

//重载赋值运算符

MyString & MyString::operator=(const MyString & st)

{

if (this == &st)

return \*this;

delete[] m\_str;//先释放原本的空间

m\_len = st.m\_len;

m\_str = new char[m\_len + 1];//再开辟适合的空间

//std::strcpy(m\_str, st.m\_str);

strcpy\_s(m\_str, m\_len + 1, st.m\_str);//拷贝

return \*this;

}

## 【特殊成员函数②】

【移动构造函数】move constructor

【移动赋值运算符】move assignment operator

# 【类型转换】

## 【标准类型】

【隐式转换】

long num = 8; //int转换为long

double price = 11; //int转换为double

int side = 3.333; //double转换为int

隐式转换是自动进行的

【自动转换】何时发生

①一种类型赋值给另一种类型

②表达式中包含不同类型

③将参数传递给函数

【自动转换可能发生的问题】

|  |  |
| --- | --- |
| 大浮点转小浮点（double→float） | 精度降低（超出取值范围则结果不确定） |
| 浮点转整形（float→int） | 丢失小数部分（超出取值范围则结果不确定） |
| 大整形转小整形（long→int） | 丢失域，超出取值范围则只复制右边字节 |

【隐式转换与显式转换】

int a = 12.3; //隐式转换，double转int

int b = int(12.3); //显式转换，double转int

【强制类型转换】int(value)

强制类型转换不会改变变量本身的值，而是产生一个新的可以在表达式中使用的值。

【4个强制类型转换运算符】

static\_cast<int>(variable)

【整型提升】int是最自然的类型，操作速度最快。

【缩窄转换】初始化时不允许缩窄转换

## 【类类型】

### 【标准类型→类类型】

【隐式转换】

只有接受一个参数的构造函数才能作为转换函数，它定义了从参数类型到类类型的转换。

下面定义了一个简单的类，它的构造函数只有一个参数

class Book{

private:

int m\_pages;

public:

Book()//默认构造函数

{

std::cout << "调用了默认构造函数 " << std::endl;

m\_pages = 0;

}

Book(const int num ) //只有一个参数的构造函数

{

std::cout << "调用了一个参数的构造函数 " << std::endl;

m\_pages = num;

}

};

在main函数中

Book bk; //默认构造函数

bk = 100; //int转换为Cake类型

bk = 12.3; //double→int，int→Cake



第二行发生了隐式类型转换，调用了只有一个参数的构造函数作为转换函数。只有一个参数的构造函数定义了从参数类型到类类型的转换。先将int类型的12转化为Cake类型的临时对象，再把临时对象复制给ck01（通过=发生的浅拷贝）。

第三行发生两次转换。

【禁止隐式转换】

如果不想发生在这种隐式转换，可以给构造函数加关键字explicit，禁止隐式转换。

explicit Book(const int num = 0);

Book bk; //默认构造函数

bk = 100; //【隐式转换】加了explicit之后就被禁止了

bk = Book(100); //【显式转换】？长得很像显式转换，实际上是构造函数

### 【类类型→标准类型】

为了实现把Book对象赋值给int

Book bk = 100;

int value = int(bk); //Book转int，显式转换

使用了operator形式的转换函数。

Book::operator int()const //从Book到int的转换函数

{

return m\_pages;

}

①转函数是成员函数，所以不需要指出参数

②int已经指出了返回值，所以不需要返回值

隐式转换如下：

int value = bk;

如果有operator int()和operator double()，根据value的类型由编译器自动选择转换函数。如果面对cout，编译器不知道要选择呢一个转换函数，出现二义性，编译器会报错。

为了禁止隐式转换，C++11中可以对转换函数加explicit。

此外，除了定义转函数operator int()，还有另外的选择是定义成员函数

Book Book\_to\_int(int argument);

## 【总结】

|  |  |  |  |
| --- | --- | --- | --- |
|  | 方法 | 隐式 | 显式 |
| 标准类型  double转int |  | int side = 3.333; | int b = int(12.3); |
| 标准类型→类  int转Book | 一个参数的构造函数作为转换函数 | Book bk = 100; | Book bk = Book(100);//？ |
| 类→标准类型  Book转int | 转换函数  operator int() | int value = bk; | int value = int(bk); |

第二行的隐式与显式（先不论这样算不算显式），都是调用了同一个构造函数（只有一个参数的构造函数）。

# 【友元】

通常，公有方法是访问类的唯一途径。但友元提供了另一种途径，通过让函数成为友元，可与赋予函数与成员函数一样的访问权限。友元有3种：

## 【友元函数】

用friend在类内声明，像普通函数一样定义（在类外）。

## 【友元类】

## 【友元成员函数】

# 【随机数】

srand()设置随机数种子

rand()产生随机数整数

#include <ctime>//time()

#include <cstdlib>//srand() rand()

srand(time(0)); //使用当前时间作为随机数种子

int degree = rand() % 360;//获得360以内的整数

# 【继承】

父类引用可以引用子类对象，而无需进行强制类型转换

# 【C++11】

## 【auto】自动推断类型

## 【范围for循环】

【目的】简化一种任务：对数组的每个元素执行相同操作（或者vector、array）

int arr[] = { 1, 2, 3, 4, 5, 6 };

for (int x : arr)

cout << x << endl;

类比于python的循环

for x in arr：

print(x)