目录

[说明 2](#_Toc55335646)

[++, --前置（后置）自增（自减）运算符的细节 2](#_Toc55335647)

[::双冒号的用法 3](#_Toc55335648)

[anonymous匿名类型 3](#_Toc55335649)

[array数组额外知识积累 3](#_Toc55335650)

[assembly内联汇编 3](#_Toc55335651)

[auto, decltype追踪返回类型 3](#_Toc55335652)

[auto关键字新定义（C++11） 4](#_Toc55335653)

[c++版本号 5](#_Toc55335654)

[cast强制类型转换 5](#_Toc55335655)

[cin, cout输入输出的常用用法 6](#_Toc55335656)

[字符串输入输出的常用用法string, char, char[] 7](#_Toc55335657)

[concurrency、multithread并发和多线程（C++11） 8](#_Toc55335658)

[constexpr常量表达式 12](#_Toc55335659)

[const、&引用、函数参数相关的知识点 12](#_Toc55335660)

[const、volatile、mutable用法总结 12](#_Toc55335661)

[convention数值的书写和各种数据类型的特殊用法 14](#_Toc55335662)

[convention命名naming与书写惯例 15](#_Toc55335663)

[convention换行续行 15](#_Toc55335664)

[CString/C字符串处理和string字符串数据转换 15](#_Toc55335665)

[decltype、typeid、动态类型和运行时类型识别 16](#_Toc55335666)

[enumeration枚举、强类型枚举 17](#_Toc55335667)

[for循环--基于范围range-based 17](#_Toc55335668)

[function函数指针、函数引用和函数名做函数参数 18](#_Toc55335669)

[function函数名称外围()括号的用法 19](#_Toc55335670)

[functional标准库和函数对象（C++11） 19](#_Toc55335671)

[initialization初始化实现细节 20](#_Toc55335672)

[initializer list初始化列表 （C++11） 20](#_Toc55335673)

[lambda函数、仿函数与编程范型 21](#_Toc55335674)

[lvalues、rvalues、（左值）引用和右值引用 23](#_Toc55335675)

[移动语义、完美转发 23](#_Toc55335676)

[main()函数参数 25](#_Toc55335677)

[nothrow常量用法 25](#_Toc55335678)

[nullptr、NULL与指针空值 25](#_Toc55335679)

[operator precedence & associativity运算符优先级与结合性 26](#_Toc55335680)

[POD类型 27](#_Toc55335681)

[preprocessing预处理 28](#_Toc55335682)

[preprocessor macro预定义宏和预定义标识符（C99，C++11兼容） 28](#_Toc55335683)

[preprocessor directives预处理命令和\_Pragma操作符 28](#_Toc55335684)

[printf与格式化输出 29](#_Toc55335685)

[RNG随机数生成器 31](#_Toc55335686)

[smart pointer智能指针 31](#_Toc55335687)

[static关键字的用法和含义以及global全局变量 32](#_Toc55335688)

[stream manipulator流操作符改变cin/cout格式 32](#_Toc55335689)

[struct结构体 33](#_Toc55335690)

[template模板 33](#_Toc55335691)

[typename关键字的用法、name lookup名称解析 34](#_Toc55335692)

[variadic function变长函数、变长宏和变长模板 35](#_Toc55335693)

[typedef用法 36](#_Toc55335694)

[最小垃圾回收 36](#_Toc55335695)

# 说明

本笔记用于记录c++课程额外的积累知识；同时还有c++11版本更新的内容

# ++, --前置（后置）自增（自减）运算符的细节

前置自增pre-increment，前置自减pre-decrement

后置自增post-increment，后置自减post-decrement

1. 返回类型不同

前置++的返回类型是左值引用，后置++的返回类型const右值。而左值和右值，决定了前置++和后置++的用法。++a的返回类型为什么是引用呢？这样做的原因应该就是：与内置类型的行为保持一致。前置++返回的总是被自增的对象本身。因此，++(++a)的效果就是a被自增两次。

2. 形参的区别

前置++没有形参，而后置++有一个int形参，但是该形参也没有被用到。很奇怪，难道有什么特殊的用意？其实也没有特殊的用意，只是为了绕过语法的限制。

前置++与后置++的操作符重载函数，函数原型必须不同。否则就违反了“重载函数必须拥有不同的函数原型”的语法规定。虽然前置++与后置++的返回类型不同，但是返回类型不属于函数原型。为了绕过语法限制，只好给后置++增加了一个int形参。原因就是这么简单，真的没其他特殊用意。

在自定义类型重载自增自减运算符时，只要没有参数便被编译器视为前置，只要有参数，无论是double还是int都认为是后置。另外，重载函数须为成员函数。

3. 代码实现的区别

前置++的实现比较简单，自增之后，将\*this返回即可。需要注意的是，一定要返回\*this。

后置++的实现稍微麻烦一些。因为要返回自增之前的对象，所以先将对象拷贝一份，再进行自增，最后返回那个拷贝。

4. 效率的区别

如果不需要返回自增之前的值，那么前置++和后置++的计算效果都一样。但是，我们仍然应该优先使用前置++，尤其是对于用户自定义类型的自增操作。前置++的效率更高，理由是：后置++会生成临时对象。

C++Primer中（P132）有这样简介的描述：

前置版本将对象本身作为左值返回，后置版本则将原始对象的副本作为右值返回，两种运算符必须作用于左值运算对象。后置版本需要拷贝副本，所以会影响程序的性能。

# ::双冒号的用法

:: 是作用域符，是运算符中等级最高的，它分为三种:

1)global scope(全局作用域符），用法（::name)

如在程序中的某一处你想调用全局变量a，那么就写成::a；（也可以是全局函数）

2)class scope(类作用域符），用法(class::name)

如果想调用class A中的成员变量a，那么就写成A::a

3)namespace scope(命名空间作用域符），用法(namespace::name)

如果想调用namespace std中的cout成员，你就写成std::cout（相当于using namespace std；cout）意思是在这里我想用cout对象是命名空间std中的cout（即就是标准库里边的cout）。

他们都是左关联（left-associativity)，他们的作用都是为了更明确的调用你想要的变量。

# anonymous匿名类型

很多时候，自定义类型、结构体、枚举等广泛意义的“类”可以在声明时不加名称，即匿名；匿名的类型可以作为临时的数据结构使用，但是显然其存在的意义便是不希望这个数据类型被广泛运用。

# array数组额外知识积累

1.初始化

数组的初始化可以直接用初始化列表进行，如int a[5] = {1,2,3,4,5}。如果给出的具体数值不够用，则用0补充。注意，如果不进行初始化则不会自动全赋0。

2.多维数组

多维数组作为函数参数时，除了第一个下标，其余下标都必须指定。实际上，数组变量的本质就是指针，n维数组作参数，实际上就是以指向确定尺寸的n-1维数组的指针为参数。

# assembly内联汇编

内联汇编使用asm或\_\_asm或\_\_asm\_\_关键字（实际上是各种宏，效果一样，无作用上的区别），用以说明其后的代码为内联汇编代码。

# auto, decltype追踪返回类型

1.定义

追踪返回类型是和auto与decltype类型推导共同使用的为函数泛型编程提供基础的机制。即在函数模板的使用中，对函数的返回类型采用推导的方式决定，而推导则依据函数类型之后声明的函数参数类型，因此要使用追踪返回类型的方式来定义模板，以避免声明前使用参数类型的问题。

举例：一个泛型求和函数模板

template<typename T1, typename T2>

auto Sum(T1 & t1, T2 & t2) -> decltype(t1 + t2){

return t1 + t2;}

•auto占位符和->return\_type是构成追踪返回类型函数的两个基本元素，可以简单理解为auto就是为return\_type占位所用，等价于将后者替换auto所使用的普通声明方式

2.使用方法

追踪返回类型其实是一种定义函数的方法，和普通的函数声明在一般情况下可以通用。当然其存在的意义就在于普通声明方式失效的泛型编程环境下。编写出的函数具备泛型使用能力，并且代码及其简化。

①举例：通过嵌套实现代码简化

int (\*(\*pf())())() {

return nullptr;}

auto pf1() -> auto (\*)() -> int (\*)() {

return nullptr;} //两个函数pf和pf1完全一样

•用<type\_traits>中，is\_same<decltype(pf), decltype(pf1)>::value可以知道二者的类型是否一致

②举例：转发函数

template <class T>

auto Forward(T t) -> decltype(f(t)){

return f(t);}

③没有返回值的函数也可以使用追踪返回类型，同样声明为void即可

# auto关键字新定义（C++11）

以往的auto是存储类型指示符（storage-class-specifier，如static、extern、thread\_ local等都是存储类型指示符），表示的是自动存储期的局部变量（一般都是是默认的）。C++11中，作为一个新的类型指示符（type-specifier，如int、float等都是类型指示符）用来指示编译器进行类型推导。

1.auto的用法

auto实际上是一个类型声明时的“占位符”，编译器会在编译时期将其替代为推导出的变量实际类型。为了让编译器能够有推导的基础，auto声明的变量必须被初始化。

auto的使用方法和一般的类型指示符一样，如int，float等。特别的，auto可以new（堆变量运算符）一起使用，如auto p = new auto (1)

2.auto的用处

•对于包含STL的类型名，往往很长，比如std::vector<std::string>::iterator等，可以使用auto来让编译器自动推导。

3.auto使用细则

①对于指针类型的推导，使用auto或auto\*没有区别，例：

int x;

int \*y = &x;

auto \*a = &x;

auto c = y;

auto \*d = y; //a，c，d三个变量都为指针，且都指向x，即完全一样

②如果要使得auto声明的变量是另一个变量的引用，则必须使用auto&

③用auto声明的变量，没法从其初始化表达式（或变量）中继承cv限制符（const或volatile），只有声明引用时（auto&）才可以。也就是说，用来初始化的变量如果带有const属性，那么auto定义的变量并不继承const属性。

④用auto声明多个变量时，所有变量对应的auto只能取一种类型。实际上，只有第一个变量用于类型推导，确定auto的取型，其他的变量和其共享同一类型的auto。在这里，auto必须理解为一个占位符（placeholder）。例：

auto x = 1, y = 2;

const auto\* m = &x, n=1;

auto o = 1, &p = o, \*q = &p; //这些写法都合法，但是为了不易出错，应该分开定义

⑤auto在以下几种情况不能使用，无法通过编译：

void fun（auto x = 1）{} //auto不能作为形参，如果想用泛型函数，必须用模板

struct str

{

auto var = 10; //非静态成员变量，不能用auto

};

char x[3];

auto y = x; //可以用auto推导数组类型

auto z[3] = x; //不能用auto直接定义数组

vector<auto> v = {1}; //不能用auto作模板参数

⑥c++11中的auto和之前版本的auto不兼容，即c++11中只保留了类型指示符的功能。

# c++版本号

c++98是第一版c++标准。

c++03主要是98版的修订，只增加了一个新内容——value initialization。

c++0x是c++11标准在编撰过程中的名称，因为原定完成日期是08年。

c++11是2011年发布的标准，增加了大量新内容和改进。

c++14是2014年的标准，针对11标准的修订。

c++17是现行标准（2020）。

# cast强制类型转换

\* C++提供了四种新的类型强制：

static\_cast

const\_cast

reinterpret\_cast

dynamic\_cast

1、staic\_cast静态强制

不能在无关的指针之间进行static类型强制

class CAnimal

{

public:

CAnimal(){}

};

class CGiraffe:public CAnimal

{

public:

CGiraffe(){}

};

int main(void)

{

CAnimal an;

CGiraffe jean;

an = static\_cast<CAnimal>(jean);//将对象jean强制成CAnimal类型

return 0;

}

2、const\_cast类型强制

const\_cast类型强制将一个const变量变成一个非const的等价形式

int main()

{

const int j = 99;

int \* k;

k = const\_cast<int \*>(&j);//解除const

return 0;

}

3、reinterpret\_cast运算符

reinterpret\_cast运算符用来将一个类型指针转变为另一种类型的指针，也用在将整型量转为指针，或将指针转为整型量上；

int main()

{

int j = 10;

int \* ptr = &j;

char \* cptr;

cptr = reinterpret\_cast<char \*>(ptr);//将int指针类型转变为char的指针类型

return 0;

}

4、dynamic\_cast运算符

dynamic\_cast的主要目的是：

1）它返回派生类对象的地址；

2）它测试基类指针是否指向下一尖括号<>中所指定类型的对象

也称为down casting向下转换。dynamic\_cast是一个运行时类型信息，dynamic\_cast运算符将指向派生对象的基类部分的基类指针转变为指向派生对象的派生类指针，dynamic\_cast必须严格地指定与派生对象相同的类，或者它返回NULL指针;

class CAnimal { };

class CGiraffe:public CAnimal { };

class CGoat:public CAnimal { };

int main()

{

CGiraffe gene;

CAnimal \* aptr = &gene;

CGiraffe \* ptr1,\* ptr2;

ptr1 = dynamic\_cast<CGiraffe \*>(aptr);

ptr2 = dynamic\_cast<CGoat \*>(aptr); //return NULL

return 0;

}

# cin, cout输入输出的常用用法

cin的输入错误处理

当遇到无效字符（输入字符和接受类型不匹配）或遇到文件结束符EOF时（EOF不是换行符，是一个因平台而定的特殊标志，windows是Ctrl+Z，键盘输出后在控制台显示成^Z），cin将处于出错状态，之后对于其的所有提取操作将终止，直到使用cin.clear()去除该状态（一般还需要调用ignore函数来跳过原本出错的输入以便重输）。正常状态good下，cin的值为true，其他所有出错状态时均为0。

进一步而言，cin的不同状态有具体的标志和查询函数，见下：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **iostate value (member constants)** | **indicates** | **functions to check state flags** | | | | |
| **good()** | **eof()** | **fail()** | **bad()** | **rdstate()** |
| **goodbit** | No errors (zero value iostate) | true | false | false | false | goodbit |
| **eofbit** | End-of-File reached on input operation | false | true | false | false | eofbit |
| **failbit** | Logical error on i/o operation | false | false | true | false | failbit |
| **badbit** | Read/writing error on i/o operation | false | false | true | true | badbit |

eofbit, failbit and badbit are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator). goodbit is all 0.

## 字符串输入输出的常用用法string, char, char[]

std::string类是C++引入的类，需要<string>。C-String实际上可用字符数组char[]实现，外加’\0’作为结尾，而string类中的字符不包括’\0’。

对于字符串类型string的完整详解见《C++常用函数和类知识整理》

1. 读入字符串对象

·读入无空格的字符串cin >> str;

·读入一行字符串，可能包含空格getline(cin, str);

2. 读入字符

·读入非空字符cin >> ch;

·读入任何字符cin.get(ch); 或 ch=cin.get();

·跳过输入缓冲中n个字符，除非遇到c字符 cin.ignore(n, c);

常用于处理输入中的多余或无效字符。

·跳过输入缓冲中下一个字符cin.ignore();

3. 读入C-string或字符数组

·读入无空格的字符数组，Cstr可能会不够用 cin >> Cstr;

·读入定额的字符 cin.getline(Cstr, SIZE);

4. 初始化和赋值C-String或字符数组

·可以如string一样由字符串常量初始化

·但是后续赋值或复制必须用strcpy等函数。

5. 字符检测

需要<cctype>

·isalpha(arg): true if arg. is a letter, false otherwise

·isalnum(arg): true if arg. is a letter or digit, false otherwise

·isdigit(arg): true if arg. is a digit 0 9, false otherwise

·islower(arg): true if arg. is lowercase letter, false otherwise

·isprint(arg): true if arg. is a printable character, false otherwise

·ispunct(arg): true if arg. is a punctuation character, false otherwise

·isupper(arg): true if arg. is an uppercase letter, false otherwise

·isspace(arg): true if arg. is a whitespace character, false otherwise

6. 字符处理

需要<cctype>

toupper(arg): 将arg转化为uppercase

tolower(arg): 类似上

# concurrency、multithread并发和多线程（C++11）

1.与 C++11 多线程相关的头文件

C++11 新标准中引入了5个头文件来支持多线程编程，他们分别是<atomic> ,<thread>,<mutex>,<condition\_variable>和<future>。

·<atomic>：该头文件主要声明了两个类, std::atomic 和 std::atomic\_flag，另外还声明了一套 C 风格的原子类型和与 C 兼容的原子操作的函数。

·<thread>：该头文件主要声明了 std::thread 类，另外 std::this\_thread 命名空间也在该头文件中。

·<mutex>：该头文件主要声明了与互斥量(mutex)相关的类，包括 std::mutex 系列类，std::lock\_guard, std::unique\_lock, 以及其他的类型和函数。

·<condition\_variable>：该头文件主要声明了与条件变量相关的类，包括 std::condition\_variable 和 std::condition\_variable\_any。

·<future>：该头文件主要声明了 std::promise, std::package\_task 两个 Provider 类，以及 std::future 和 std::shared\_future 两个 Future 类，另外还有一些与之相关的类型和函数，std::async() 函数就声明在此头文件中。

2.std::thread详解

std::thread 在 <thread> 头文件中声明，因此使用 std::thread 时需要包含 <thread> 头文件。

①std::thread 构造

|  |  |
| --- | --- |
| default (1) | thread() noexcept; |
| initialization (2) | template <class Fn, class... Args>  explicit thread (Fn&& fn, Args&&... args); |
| copy [deleted] (3) | thread (const thread&) = delete; |
| move (4) | thread (thread&& x) noexcept; |

(1). 默认构造函数，创建一个空的 thread 执行对象。

(2). 初始化构造函数，创建一个 thread对象，该 thread对象可被 joinable，新产生的线程会调用 fn 函数，该函数的参数由 args 给出。

(3). 拷贝构造函数(被禁用)，意味着 thread 不可被拷贝构造。

(4). move 构造函数，move 构造函数，调用成功之后 x 不代表任何 thread 执行对象。

注意：可被 joinable 的 thread 对象必须在他们销毁之前被主线程 join 或者将其设置为 detached.

②move 赋值操作

|  |  |
| --- | --- |
| move (1) | thread& operator= (thread&& rhs) noexcept; |
| copy [deleted] (2) | thread& operator= (const thread&) = delete; |

(1). move 赋值操作，如果当前对象不可 joinable，需要传递一个右值引用(rhs)给 move 赋值操作；如果当前对象可被 joinable，则 terminate() 报错。

(2). 拷贝赋值操作被禁用，thread 对象不可被拷贝。

③其他成员函数

get\_id——获取线程 ID。

joinable——检查线程是否可被 join。

join——Join 线程。

detach——Detach 线程

swap——Swap 线程 。

native\_handle——返回 native handle。

hardware\_concurrency [static]——检测硬件并发特性。

3.std::mutex详解

Mutex 又称互斥量，C++ 11中与 Mutex 相关的类（包括锁类型）和函数都声明在 <mutex> 头文件中，所以如果你需要使用 std::mutex，就必须包含 <mutex> 头文件。

①<mutex> 头文件介绍

·Mutex 系列类(四种)

std::mutex，最基本的 Mutex 类。

std::recursive\_mutex，递归 Mutex 类。

std::time\_mutex，定时 Mutex 类。

std::recursive\_timed\_mutex，定时递归 Mutex 类。

·Lock 类（两种）

std::lock\_guard，与 Mutex RAII 相关，方便线程对互斥量上锁。

std::unique\_lock，与 Mutex RAII 相关，方便线程对互斥量上锁，但提供了更好的上锁和解锁控制。

·其他类型

std::once\_flag

std::adopt\_lock\_t

std::defer\_lock\_t

std::try\_to\_lock\_t

·函数

std::try\_lock，尝试同时对多个互斥量上锁。

std::lock，可以同时对多个互斥量上锁。

std::call\_once，如果多个线程需要同时调用某个函数，call\_once 可以保证多个线程对该函数只调用一次。

②std::mutex 介绍

下面以 std::mutex 为例介绍 C++11 中的互斥量用法。std::mutex 是C++11 中最基本的互斥量，std::mutex 对象提供了独占所有权的特性——即不支持递归地对 std::mutex 对象上锁，而 std::recursive\_lock 则可以递归地对互斥量对象上锁。

③std::mutex 的成员函数

mutex()，构造函数，std::mutex不允许拷贝构造，也不允许 move 拷贝，最初产生的 mutex 对象是处于 unlocked 状态的。

lock()，调用线程将锁住该互斥量。线程调用该函数会发生下面 3 种情况：(1). 如果该互斥量当前没有被锁住，则调用线程将该互斥量锁住，直到调用 unlock之前，该线程一直拥有该锁。(2). 如果当前互斥量被其他线程锁住，则当前的调用线程被阻塞住。(3). 如果当前互斥量被当前调用线程锁住，则会产生死锁(deadlock)。

unlock()， 解锁，释放对互斥量的所有权。

try\_lock()，尝试锁住互斥量，如果互斥量被其他线程占有，则当前线程也不会被阻塞。线程调用该函数也会出现下面 3 种情况，(1). 如果当前互斥量没有被其他线程占有，则该线程锁住互斥量，直到该线程调用 unlock 释放互斥量。(2). 如果当前互斥量被其他线程锁住，则当前调用线程返回 false，而并不会被阻塞掉。(3). 如果当前互斥量被当前调用线程锁住，则会产生死锁(deadlock)。

④std::recursive\_mutex 介绍

std::recursive\_mutex 与 std::mutex 一样，也是一种可以被上锁的对象，但是和 std::mutex 不同的是，std::recursive\_mutex 允许同一个线程对互斥量多次上锁（即递归上锁），来获得对互斥量对象的多层所有权，std::recursive\_mutex 释放互斥量时需要调用与该锁层次深度相同次数的 unlock()，可理解为 lock() 次数和 unlock() 次数相同，除此之外，std::recursive\_mutex 的特性和 std::mutex 大致相同。

⑤std::time\_mutex 介绍

std::time\_mutex 比 std::mutex 多了两个成员函数，try\_lock\_for()，try\_lock\_until()。

try\_lock\_for 函数接受一个时间范围，表示在这一段时间范围之内线程如果没有获得锁则被阻塞住（与 std::mutex 的 try\_lock() 不同，try\_lock 如果被调用时没有获得锁则直接返回 false），如果在此期间其他线程释放了锁，则该线程可以获得对互斥量的锁，如果超时（即在指定时间内还是没有获得锁），则返回 false。

try\_lock\_until 函数则接受一个时间点作为参数，在指定时间点未到来之前线程如果没有获得锁则被阻塞住，如果在此期间其他线程释放了锁，则该线程可以获得对互斥量的锁，如果超时（即在指定时间内还是没有获得锁），则返回 false。

⑥std::recursive\_timed\_mutex 介绍

和 std:recursive\_mutex 与 std::mutex 的关系一样，std::recursive\_timed\_mutex 的特性也可以从 std::timed\_mutex 推导出来

⑦std::lock\_guard<class Mutex>

本身是一个模板类，其对象管理一个mutex-like类型的对象，确保它一直被当前线程locked，直到析构时自动unlock。

⑧std::unique\_lock

4.<atomic>原子类型的使用

原子类型是一种支持原子操作（多线程中不能并行执行的最小指令集合）的数据类型，库中已定义有和所有内置基本数据类型对应的原子类型（仅限各种整型），但更为通用的是使用atomic类模板来任意定义出需要的原子类型。在原子类型的数据上进行的操作被编译器自动实现为原子操作，这是较为简单的实现线程安全和同步的方式。

例：std::atomic<T> t {构造参数};

注意：原子类型数据不能从拷贝构造、移动构造和赋值=构造中获得。但是用原子类型数据拷贝、赋值构造对应的内置数据类型是可以的。

①atomic类型的操作

由于底层细粒度的互斥实现是平台相关的，因此atomic类型中将平台相关的内容封装成了统一的操作接口，因此这些操作也就是原子操作。不同的atomic具体类型支持的操作接口也不完全一致，如下表。对于操作接口的具体功能，需另行查找。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 操作 | atomic\_flag | atomic\_bool | atomic-integral-type | atomic<bool> | atomic<T\*> | atomic<integral-type> | atomic<class-type> |
| test\_and\_set | y |  |  |  |  |  |  |
| clear | y |  |  |  |  |  |  |
| is\_lock\_free |  | y | y | y | y | y | y |
| load |  | y | y | y | y | y | y |
| store |  | y | y | y | y | y | y |
| exchange |  | y | y | y | y | y | y |
| compare\_exchange\_weak, compare\_exchange\_strong |  | y | y | y | y | y | y |
| fetch\_add, += |  |  | y |  | y | y |  |
| fetch\_sub, -= |  |  | y |  | y | y |  |
| fetch\_or, |= |  | y |  |  | y |  |  |
| fetch\_and, &= |  |  | y |  |  | y |  |
| fetch\_xor, ^= |  |  | y |  |  | y |  |
| ++, -- |  |  | y |  | y | y | y |

注意：

·atomic\_flag不同于atomic\_bool，其本身是无锁的。并且由其支持的特殊接口可见其特殊的用法。它可以用于实现一个自旋锁spin lock，也可以封装成一个锁操作。

·该表中没有列出全部的重载运算符函数。

5.内存模型memory model，顺序一致性sequential consistent与memory\_order

本节讨论是针对在多线程环境下，某些编译器和CPU为了优化代码，会进行代码重排，可能会对多线程间共享的数据产生违背程序目标的作用。即针对多线程环境下，代码优化需要加入新的策略。

内存模型是指如何规定代码对于内存的读写顺序。进一步而言，内存模型可以分为硬件平台层面与编译器的软件层面，因为在不同的实际系统环境下，CPU和编译器都可以打乱代码顺序。CPU打乱的是机器指令顺序以满足流水线化，编译器可以打乱源代码或机器指令顺序以方便CPU的工作，也可以做到加入特殊指令（内存栅栏memory barrier）以控制CPU打乱机器代码的行为。

顺序一致性，指代码在线程中运行的顺序和程序源代码指示的一致。这是根本不进行代码顺序优化的最简单的也是最低效的方式。无论何时都遵循顺序一致来执行代码的方式称为强顺序strong ordered内存模型，对应硬件平台有x86和SPARC（TSO模式）。反之，称为弱顺序内存模型，如Alpha、PowerPC、Itanlium、ArmV7。

memory\_order内存顺序是C++11中给予程序员控制编译器对原子操作进行/不进行代码重排的功能。默认情况下，编译器会禁用对原子操作的重排优化，若平台是弱顺序的，编译器也会加入内存栅栏阻止CPU处打乱顺序。

①memory\_order使用方法

在对原子类型数据进行操作时，调用其成员函数来实现操作即可传递memory\_order的枚举值来设置。所有枚举值和功能如下：

|  |  |
| --- | --- |
| memory\_order\_relaxed | 不对执行顺序做任何保证 |
| memory\_order\_acquire | 本线程中，所有后续的读操作必须在本条原子操作完成后执行 |
| memory\_order\_release | 本线程中，所有之前的写操作完成后才能执行本条原子操作 |
| memory\_order\_acq\_rel | 同时包含acquire和release标记 |
| memory\_order\_consume | 本线程中，所有后续的有关本原子类型的操作，必须在本条原子操作完成之后执行（不一定是相同数据，是相同原子类型） |
| memory\_order\_seq\_cst | 全部存取都按顺序执行 |

·memory\_order\_seq\_cst即是顺序一致的，也是默认选项。因此不传memory\_order参数或使用运算符都是默认顺序一致的。

·不同atomic成员函数可以使用的memory\_order是不同的：

store存储可使用relaxed、release、seq\_cst

load读取可使用relaxed、consume、acquire、seq\_cst

read-modify-write同时读写可使用realxed、consume、acquire、release、acq\_rel、seq\_cst。

6.线程局部存储thread local storage，TLS

一般情况下，线程会拥有自己的栈空间，但是堆空间、静态数据区（data、bss段）是共享的。对于这样的一个问题——线程如何管理自身的全局数据，即不会和其他线程共享且在自身线程内为全局，C++11没有给出实现标准，只有定义TLS变量的语法。

int thread\_local xxx;

对于TLS变量的运行前静态分配和运行时动态分配，二者性能不一样，需要平台相关信息才能确定其采用何种方式。

# constexpr常量表达式

constexpr的主要功能是定义编译时常量性，和const不同的是，后者用于定义运行时才确定的常量性。也可以说constexpr修饰的内容编译时可定，const修饰的内容编译时不一定。

1.常量表达式函数

·可声明为常量表达式函数的要求：

①函数体只有单一的return语句——可以有一些不会产生实际代码的语句，如using，typedef

②函数必须返回值

③使用前必须已有定义——光有函数原型声明是不够的

④return的数据不能使用非常量表达式的函数和全局数据——仍然是编译时和运行时常量性的区别问题，但注意常量表达式成员函数可以返回成员变量，只要构造函数是constexpr修饰的，即该类是constexpr的。

注意：C++11不许常量表达式作用于virtual成员函数，因为后者是运行时行为相关的。

例子：

constexpr int h() { return 1; }

2.常量表达式值constant-expression value

内置数据类型可以直接使用constexpr修饰，自定义类型必须声明constexpr修饰的构造函数，才能使用constexpr声明对象。另一方面，constexpr自定义类型是通过constexpr构造函数定义的，而不是constexpr直接修饰类名。

·常量表达式值的构造函数要求：

①函数体必须为空

②初始化列表只能由常量表达式赋值——构造函数声明中仍可以使用普通类型的参数，但实例化时必须使用常量表达式

3.常量表达式的其他应用

·constexpr可以用于修饰模板函数，对于某些调用满足constexpr条件则修饰生效，反之某些函数调用若不满足条件，则自动忽略修饰。

·constexpr修饰的函数可以通过递归（c++11标准规定编译器最少应支持512层）来实现元编程meta programming，即通过一定技巧让编译器在编译阶段就执行设计好的程序逻辑。在此之前有模板元编程的技术，但是constexpr元编程更强大。

注意：低级版本的编译器可能不会完全实现constexpr的编译时计算功能。尤其是复杂的constexpr函数递归计算。

# const、&引用、函数参数相关的知识点

·函数的默认参数声明必须在参数列表的最后，同理调用函数时省略的参数也必须在最后，

不能将默认参数和非默认参数顺序交杂。

## const、volatile、mutable用法总结

0.函数参数知识概要

抛开函数体本身的功能，函数运行前后有两个参数/返回值传递的步骤，可以理解参数传递和返回值传递的过程是类似的，都可以用值传递、指针和引用传递的方式，而形参实参的传递过程类似于return后表达式和函数返回值的传递过程，同样有着拷贝、引用、移用的方式。

1.const修饰函数参数

const只能修饰输入参数，输出参数的不能用const保护。这在任何传递方式下都一样。

如果是“指针传递”输入参数，则可以防止意外改动该指针所指向的内容。

“引用传递”输入参数，使用const可以防止改动实参的值。

如果是“值传递”输入参数，形参是实参的一个拷贝，根本不会改动实参的值，const修饰没有意义。

另外一个方面，“值传递”需要拷贝参数或返回值，对于用户自定义类型更加消耗时间，“指针传递”或“引用传递”参数效率更高，不过对于内部数据类型效率差别不明显。

•再进一步说明，对于指针传递的情况，指针也是变量或者参数，对于指针变量而言，仍然是值传递

2.cosnt修饰函数返回值

“指针传递”方式的函数返回值如果加const修饰，即返回的指针指向的内容不能修改，如果要将返回值赋给其他指针，那么其他指针也必须为const型。

“值传递”函数一般的返回值是返回值变量的一个数值拷贝，因此加const修饰没有意义。

“引用传递”函数返回的是返回值变量的引用，即别名，加const修饰后返回的变量不能修改。

3.const类成员/函数和const类/对象

详见“C++面向对象程序设计笔记”

4.const修饰普通变量或指针

const修饰变量，一般有两种写法：

const TYPE value;

TYPE const value;

这两种写法在本质上是一样的。它的含义是：const修饰的类型为TYPE的变量value是不可变的。对于一个非指针的类型TYPE，无论怎么写，都是一个含义，即value值不可变。 但是对于指针类型的TYPE，不同的写法会有不同情况：

①指针本身是常量不可变

(char\*) const pContent;

②指针所指向的内容是常量不可变

const (char) \*pContent;

(char) const \*pContent;

③两者都不可变

const char\* const pContent;

5.mutable的作用

mutable的中文意思是“可变的，易变的”，跟constant（即C++中的const）是反义词。在C++中，mutable也是为了突破const的限制而设置的。被mutable修饰的变量(mutable只能由于修饰类的非静态数据成员)，将永远处于可变的状态，即使在一个const函数中。我们知道，假如类的成员函数不会改变对象的状态，那么这个成员函数一般会声明为const。但是，有些时候，我们需要在const的函数里面修改一些跟类状态无关的数据成员，那么这个数据成员就应该被mutable来修饰。详见“C++面向对象程序设计笔记”。

6.volatile的作用

volatile的本意是“易变的”,volatile关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比如操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。当要求使用volatile 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被寄存。例如：

volatile int i=10;

·下面是volatile变量的几个例子：

1). 并行设备的硬件寄存器（如：状态寄存器）

2). 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)

3). 多线程应用中被几个任务共享的变量

·几个有关volatile的问题：

1). 一个参数既可以是const还可以是volatile吗？解释为什么。

2). 一个指针可以是volatile 吗？解释为什么。

3). 下面的函数有什么错误：

int square(volatile int \*ptr)

{

return \*ptr \* \*ptr;

}

下面是答案：

1). 是的。一个例子是只读的状态寄存器。它是volatile因为它可能被意想不到地改变。它是const因为程序不应该试图去修改它。

2). 是的。尽管这并不很常见。一个例子是当一个中服务子程序修该一个指向一个buffer的指针时。

3). 这段代码的有个恶作剧。这段代码的目的是用来返指针ptr指向值的平方，但是，由于ptr指向一个volatile型参数，编译器将产生类似下面的代码：

int square(volatile int \*ptr)

{

int a,b;

a = \*ptr;

b = \*ptr;

return a \* b;

}

由于\*ptr的值可能被意想不到地该变，因此a和b可能是不同的。结果，这段代码可能返不是你所期望的平方值！正确的代码如下：

long square(volatile int \*ptr)

{

int a;

a = \*ptr;

return a \* a;

}

# convention数值的书写和各种数据类型的特殊用法

1. integer整数

·所有整型类型除了int本身外，其他的int关键词均可省略

·在整数的后面加L表示long int，LL表示long long int，如234L，234LL

·整数前加0表示该数为八进制（不要将该0视为无效或错误），加0x表示16进制。

2. floating-point浮点数

·如果使用自动类型auto，默认是采用double类型

·所有浮点数类型均带有符号

·在浮点数后加F表示单精度，L表示扩展双精度

·浮点数的表示方式有些特别，所以直接比较两个浮点数是否相等不可取。两个浮点数仍可比较大小，或者相减与0比较。

# convention命名naming与书写惯例

命名和书写惯例的选择无所谓但是必须贯彻整个程序：

1. 大小写法

变量和函数：

首词小写，后续各词首字母大写，各个词之间直接相连（骆驼命名法Camel case Notation）

类名、属性和命名空间：

所有词首字母大写（大驼峰法Upper Camel Case或帕斯卡命名法Pascal Case Notation）

·Camel Case和Pascal Case在不同的编程语言中惯例不同。上面的是较通用的共识，用于C++和Java，而非C#。

2. 下划线法

变量：

使用下划线分隔名称中的词，但是所有词均小写

3. 匈牙利命名法Hungarian notation

变量：

基本原则是，变量名=属性+类型+对象描述，具体属性和类型的缩写可以具体查阅。属性和类型部分全部小写，对象描述或对象本名所有单词首字母大写以区分。

# convention换行续行

C++中标准的续行符是“\”，即反斜杠放在语句末尾被断处。对于字符串，可以任意换行，只要被断处用引号封口即可；另外也可以在逗号之后进行自由换行。

# CString/C字符串处理和string字符串数据转换

null character 空字符，可表示为以下形式'\0'，NULL，int 0，在CString中是’\0’。

1. 处理CString的函数

·需要<cstring>头文件

int strlen(char \*str) // 返回字符串字符数目，不包括尾部空字符

void strcat(char \*dest, char \*source) // 将source接在dest后面，须自行注意dest界限问题

void strcpy(char \*dest, char \*source) // 将source拷贝到dest，须自行注意dest界限问题

int strcmp(char \*str1, char \* str2) // 返回值<0，str1字母序在前；若>0，str2字母序在前，若=0，则相同。注意，大小写敏感。

·注：C11标准新增strcat\_s等名称后加\_s版本的函数，这些函数更加安全易用（如执行边界检查等）。restrict是C99中的关键字，限制声明的指针为其指向对象的唯一访问途径。

errno\_t strcat\_s(char \*restrict dest, rsize\_t destsz, const char \*restrict src) //destsz标明最大可写空间，也可以省略destsz，此时调用该函数的一个模板版本，destsz相当于默认取作dest的最大空间。

2. 处理字符串与数字转换

①流处理

·需要<sstream>头文件

istringstream

ostringstream

②函数处理

to\_string(x) // 将x转换为string

stoi(const string& str, size\_t\* pos = 0, int base = 10) // 将str转换为int

stol() // 同上，转换为long

stof() // float

stod() // double

# decltype、typeid、动态类型和运行时类型识别

1. 运行时类型识别（RTTI）

c++98中对于动态类型的支持，RTTI的机制是为每个类型产生一个 type\_ info类型的数据， 程序员可以在程序中使用typeid随时查询一个变量的类型，typeid就会返回变量相应的type\_ info数据。而type\_ info的name成员函数可以返回类型的名字。而在C++ 11中，又增加了hash\_ code这个成员函数，返回该类型唯一的哈希值，以供程序员对变量的类型随时进行比较。

运行时类型识别在运行时有额外的开销，而且为时过晚，应该在编译时期就确定出类型，使用的价值大于识别。

举例：

typeid(a).name() //返回变量a的type名称

typeid(a).hash\_code() //返回变量a的类型的唯一的哈希值

2. decltype应用

①与typedef/using合用

举例：

using size\_t = decltype(sizeof(0));

using ptrdiff\_t = decltype((int\*)0 - (int\*)0);

typedef decltype(vec.begin()) vectype;

•由变量或者表达式来推导出其数据类型，是decltype的作用，结合typedef可以将复杂的非标准类型提取并且命名重用

②标准库中的应用

<type\_traits>头文件中，result\_of<X()>::type可以用于推导函数X()的返回值类型，result\_of<X()>::type可以用作类型声明，即类似于decltype是类型声明占位符

result\_of<func()>::type f;

3. decltype的使用规则

decltype(e)

①如果e是一个没有带括号的标记符表达式(id-expression)或者类成员访问表达式，那么推导结果为实体类型T，如果e是一个重载函数，无法推导

•标记符表达式即单个标记符，变量名、函数名（不带括号和参数）和数组名（不带中括号和维数）都是单个标记符，不能包括任何运算符等

②如果e的类型是T，且是一个将亡值，那么推导类型为T&&

③如果e是一个左值或者表达式结果为左值，那么推导类型为T&

•字符串字面常量为左值，除字符串外字面常量为右值

•++i返回i的左值，i++返回右值

•((e))双括号视为左值

④其他情形，如返回纯右值，推导结果仍为T

•使用<type\_traits>中，is\_rvalue\_reference<X>::value和is\_lvalue\_reference<X>::value可以判断X的类型是不是左右值引用，X可以是decltype(…)的返回结果

4. cv限制符的继承和冗余符号

•auto类型推导无法沿用初始化变量的cv限制符，但是decltype可以沿用；但是如果是带有cv限制的对象，用decltype推导其成员时，不继承cv限制符

•冗余符号，即定义时和decltype占位符同时使用的\*和&符号遇到decltype推导的结果也带 有这些符号时的情况

举例：

decltype(e) & X = Y; //如果e为左值引用类型，那么&会被忽略，不会推导出右值引用

decltype(e) \* X = &Y; //如果e为指针类型，那么\*不会被忽略，推导结果为T\*\*类型

# enumeration枚举、强类型枚举

1.枚举（c++98）

见《C++程序设计》p222

2.传统枚举的缺点

①具名的enum类型和其成员名全局可见

即enum是非强作用域类型，不同的枚举类型中不能有同名成员，除非加命名空间限定

②隐式的将enum成员转换为整型

将成员转化为数值类型后失去了原有的一点点范围限制，使用起来非常不安全

③自行封装的成本代价高

④枚举数据的大小和类型不确定

枚举成员的底层实现是数值，传统的枚举类型让编译器自行决定底层实现的数据类型。当然在不手动定义枚举成员取值时，这个问题一般不存在，都会自动取整型为底层类型。

3.强类型枚举（strong-typed enum）/枚举类

enum class NAME {…} //默认底层类型int

enum class NAME : DATATYPE {…} //自定义底层类型，除了wchar\_t

①使用强类型枚举成员必须加枚举类名，因为枚举成员非全局；因此匿名的强枚举类型意义 不大

②强枚举成员同类之间可以比较，但是不会隐式转换，如果要和其他类数据比较，必须显式 转换

③enum class和enum struct都可以作为强枚举声明的关键字

4.c++11对传统枚举的改进

改进后的传统枚举，可以自定义底层类型，也可以使用枚举类型的作用域。不过仍然和传统的枚举兼容，即默认时由编译器指定底层类型，枚举成员的作用域仍然是全局的（只是可以加枚举类型限定）。

# for循环--基于范围range-based

①范围for循环，定义部分由“迭代变量”“冒号”和“迭代范围”组成。迭代变量可以使用引用的形式，进而可以直接使用迭代范围内的实际对象。

vector<int> v = …

for (auto & e : v) …

②能够使用范围for循环，有两个条件，必须有确定范围，即有类似begin与end函数，另外循环范围所在的类必须能够实现++和==等操作。对于标准库的容器，这些条件已经满足。

③如果遍历的数组或容器大小不确定，不能使用范围for循环。

④当遍历容器对象时，若省略&符号，则迭代对象是容器的对象或者拷贝，而不是迭代器对象或容器内的实际对象，例：

for (auto e : v) … //e的类型不是迭代器，而是v中对象的拷贝

# function函数指针、函数引用和函数名做函数参数

1. 函数指针

·函数指针是在C和C++中均有定义的功能，因此对于静态成员函数或全局函数等面向过程的实体和非静态成员函数等OOP实体的语法是不一样的。当然C++编译器向下兼容，即不加类名和&的使用方法仍被接受。

①定义形式：

返回类型 (\*函数指针名称)(参数列表) //静态成员函数或全局函数指针可不加类名

返回类型 (类名称::\*函数成员名称)(参数列表) //普通成员函数指针必须加类名

·对于const型函数的指针也必须在相同位置声明const。如void (test::\*pconst)(double)const

·构造函数和析构函数的指针没有被编译器支持（C++标准中不详）

②赋值方法：

函数指针变量 = (类名称::)函数名 //静态成员函数或全局函数指针可不加&符号

函数指针变量 = &(类名称::)函数名 //普通成员函数指针必须加&符号

③调用方法：

函数指针变量(参数列表)

(\*函数指针变量)(参数列表) //静态成员函数或全局函数指针可不加\*号

(对象名.\*函数指针变量)(参数列表) //普通成员函数指针必须加\*号，this->\*p也可以

④函数指针作为函数参数（举例）

void fun1(void (\*pfun)(int, char), int a, char c)

⑤函数指针作为返回值（举例）

void (\*fun1(double d))(int, char)

int (test::\*fun1(double d))(int, char)

// test的成员函数fun1,参数是double，返回值是test的成员函数指针int(test::\*)(int, char)

⑥函数指针数组

float (\*pfunArry[2])(float, float) = {&add, &minu};

//定义一个函数指针数组，大小为2，里面存放float (\*)(float, float)类型的指针

2.函数引用

int (&fr)()

•定义一个引用变量来引用函数名变量，用途和函数指针类似，都是为了将函数名用作函数 参数

举例：

template<typename T, typename U>

void PerfectForward(T &&t, U& Func) {

cout << t << "\tforwarded..." << endl;

Func( forward<T>(t));}

# function函数名称外围()括号的用法

这是区分宏调用与函数调用的常用方法，如(max)()，是符合C/C++标准的语法结构，它利用类函数宏与函数调用之间的一个明显区别：宏调用不属于表达式，而函数调用属于表达式（后缀表达式）。这一般常见于标准库中，如max和min等。另外也可以通过一些预处理命令禁用某些不想要的宏定义，不过这种方法显然不如上一种好，因为上一种可以同时保留两种功能，使用时不冲突。  
我们常常可以在表达式中加上括号（但宏调用不能），函数指示符加上括号后（注意要把命名空间、类名等作用域一并加入括号内），将发生从函数到指针的转换，并通过此指针调用函数；不加括号的话，函数到指针的转换被抑制，直接通过函数指示符调用函数。

# functional标准库和函数对象（C++11）

1.Callable type

Callable type 指可以像调用函数一样被调用的对象或函数，包括：

std::function

std::bind

std::result\_of

std::thread::thread

std::call\_once

std::async

std::packaged\_task

std::reference\_wrapper

根据 C++ 17 Standard，所有 Callable type 都可以通过 std::invoke 方法进行显式调用。

2.std::function

std::function类模板是一种通用的函数包装器，它可以容纳所有可以调用的对象（Callable），包括 函数、函数指针、Lambda表达式、bind表达式、成员函数及成员变量或者其他函数对象。通过 std::function 可以储存、拷贝或调用 Callable 对象。它的模板参数如下：

template< class R, class... Args >

class function<R(Args...)>

R为返回类型，Args为函数参数的类型参数包，使用时，模板参数与要存储的函数参数一致即可。

3.std::bind

顾名思义，std::bind函数用来绑定函数的某些参数并生成一个新的function对象，即对某个含有n个参数的函数进行m（m<=n）个参数的赋值，然后返回处理后的函数对象。有点类似于进行了默认参数的设置，但是不同的是被赋值的参数不再可用或可赋其他值，另外绑定赋值的参数不必在参数列表中连续。

bind用于实现偏函数（Partial Function），相当于实现了函数式编程中的 Currying（柯里化）。

比如有一函数的定义为：

void func\_muti(int a, std::string&& b, const char\* c, double d, char e) {

std::cout << a << ", " << b << ", " << c << ", " << d << ", " << e << "\n";

}

现在将此函数的一些参数绑定上值，其余部分用占位符对象（std::placeholders）表示。占位符是有序号的，代表调用此函数对象时参数在参数列表中的位置。比如：

auto f = std::bind(&func\_muti, 24, std::placeholders::\_1, "Haha", std::placeholders::\_2, 'P');

f("Hehe", 24.24);

调用这个函数对象相当于调用以下函数：

void f(std::string&& b, double d) {

std::cout << "24" << ", " << b << ", " << "Haha" << ", " << d << ", " << 'P' << "\n";

}

·需要额外注意的是，绑定成员函数时，其隐含的第一个参数是该类的实例。

# initialization初始化实现细节

C++标准/编译器中实际上对初始化有非常复杂精细的区分。每一种初始化方式并不是由源代码的形式决定的，而是编译器如何实现语义来决定的。这里列出的并非全部初始化的种类，按需学习添加即可。

0. default initialization

无initializer提供时触发

1. value initialization

initializer为空时触发

例：std::string s{};

2. direct initialization

显式提供构造函数参数

例：std::string s(“hello”);

3. copy initialization

由其他对象赋值/复制构造

例：std::string s = “hello”;

4. list initialization

用初始化列表braced-init-list来初始化单个对象

例：std::string s{‘a’, ‘b’, ‘c’};

5. aggregate initialization

用初始化列表braced-init-list来初始化集合型变量，如数组和满足某些条件的类

例：char a[3] = {‘a’, ‘b’};

6. reference initialization

用于绑定引用

例：char& c = a[0];

# initializer list初始化列表 （C++11）

①包含初始化列表情况下，可用的初始化方式，

注：只有花括号才代表c++11新增的初始化列表，其余的是c++98中已有的初始化形式， 初始化列表里可以包含左值或变量

•等号加赋值表达式（assignment-expression），如 int a=3+4

•等号加花括号的初始化列表，如int a={3+4}

•圆括号式表达式列表，如 int a(3+4)

•花括号式初始化列表，如 int a{3+4}

注：标准库中的类模板，如vector，map等，可以直接使用初始化列表，格式与一维或二维 数组相同，如vector<int> c{1,3,5}; map<int,float> d{{1,1.0f},{2,2.0f},{5,3.2f}}

②自定义类型使用初始化列表就地初始化

•自定义类中可以使用初始化列表来对所有非静态数据成员和静态常量数据成员进行就地初 始化，即类体内初始化，相当于为构造函数提供默认参数。这是c++11版本的新增内容， c++98版本不支持以上方式，对于静态非常量数据成员和c++98一样，必须在类体外初 始化。

•在构造函数或参数初始化表（和初始化列表概念不同）中对数据进行的初始化赋值后来居 上优于使用初始化列表就地初始化的“默认值”。

③自定义类型和函数使用初始化列表编程

•初始化列表，即形如{X,Y,Z…}的实体，其实是一个模板类的对象，模板类即initializer\_list<T>，T即为X,Y,Z的实际类型，所属头文件为<initialize\_list>。那么凡是定义了以这个模板类对象为参数的类型或函数都可以使用初始化列表。同样，标准模板类都定义了begin(), end()等成员函数用于迭代。

•举例：利用初始化列表初始化自定义类型

#include < vector>

#include < string>

using namespace std;

enum Gender {boy, girl};

class People {

public:

People( initializer\_ list< pair< string, Gender>> l) {

auto i = l. begin();

for (;i != l. end(); ++ i) data. push\_ back(\* i); }

private:

vector< pair< string, Gender>> data; };

People ship2012 = {{"Garfield", boy}, {"HelloKitty", girl}};

•延伸：使用初始化列表作为参数来编写自定义类型，可以利用重载运算符来实现一些功能， 如利用列表来选择一些数据进行单独或统一赋值，这是传统的数组和标准库容器不具备 的功能。

④初始化列表有可防范类型收窄的作用

•使用初始化列表，造成类型收窄的情况不会通过编译，不仅仅发出警告

•类型收窄：新类型无法表示原有类型数据的值（包括存在精度损失的情况）或转换为新类 型后再次转换为原有类型时无法获得原来的值

# lambda函数、仿函数与编程范型

1. 编程范型（Programming Paradigm）

函数式编程Functional Programming

命令式编程Imperative Programming

面向对象编程Object-oriented Programming

2. lambda函数

1）语法定义：

[capture] (parameters) mutable ->return\_type {statement}

①[capture]：捕捉列表，[]是lambda函数的引出符

②(parameters)：参数列表，类似一般的函数，不需要的话可以连同括号一起省略

③mutable：默认情况下，lambda函数总是一个const函数，mutable可以取消常量性或者省 略遵循const性；使用mutable时，参数列表不可省略（即使为空）

④->return\_type：用追踪返回类型方式声明返回值类型，无返回值时可以连同->一起省略； 在返回值类型明确的情况下，也可以省略该部分，让编译器自行对返回类型推导

⑤{statement}：函数体，与一般函数类似，除了可以使用参数外，还可以使用捕获的变量

2）使用方法：

•举例1：作为有名函数的函数体，可重用的“局部函数”

auto validate = [&] () -> bool {…};

success += validate(); //用lambda函数定义函数体，赋予validate的函数名，后可重用

•举例2：作为匿名函数就地定义和使用

const int y = [=]{

int i, val;

…}(); //用lambda函数就地定义和使用（最后有括号代表函数调用），匿名函数

3）C++11标准中的相关知识

•lambda函数是一个特殊的类型，被定义为闭包（closure）的类。每个lambda表达式产生一个闭包类型的临时对象（右值）。lambda函数类型既非函数指针类型也非自定义类型。closure类型被定义为特有的（unique）、匿名且非联合体（unnamed nonunion）的class类型。

•没有捕捉任何变量的lambda函数可以向函数指针类型转换，但是函数指针类型反过来不能向lambda类型转换。用decltype(lambda函数名)可以获得lambda函数的类型。

3. 仿函数functor/function object

lambda函数的实际实现是通过仿函数来实现的，仿函数是一个类class通过成员函数重载“()”运算符以实现类似函数的语义，实质上仿函数是一个类或结构体的成员函数。仿函数的调用即通过对象成员函数的调用实现，建立该对象时构造的数据成员即类似lambda函数的捕捉参数。仿函数的函数名是对象名。

另外值得一提的是，函数指针（即函数名）和仿函数虽然定义方法不一样，但是用法几乎一样，均常被用于函数参数进行传递，以实现调用函数内部的自定义动态逻辑，如<algorithm>中的众多函数以及某些容器类中用来设置内部顺序。虽然二者类型不同，但是往往均被模板的不定类型兼容。在<functional>中有一些预定义的仿函数模板。

4. 捕捉列表

语法上，捕捉列表由多个捕捉项组成，以逗号分隔，每一项有如下几种形式：

•[var] 表示值传递方式捕捉变量var

•[=] 表示值传递方式捕捉所有父作用域的变量（包括this）

父作用域（enclosing scope），指包含lambda函数的语句块

•[&var] 表示引用传递捕捉变量var

•[&] 表示引用传递捕捉所有父作用域的变量（包括this）

•[this] 表示值传递方式捕捉当前的this指针

•[&this] 表示引用传递方式捕捉当前的this指针

组合示例：

[=,&a,&b] 以引用传递的方式捕捉变量a和b，值传递方式捕捉其他所有变量

[&,a,this] 表示以值传递方式捕捉变量a和this，引用传递方式捕捉其他所有变量

注意事项：

•在块作用域（block scope）以外的lambda函数捕捉列表必须为空，因为不存在父作用域

•块作用域中的lambda函数只能捕捉其中的自动变量，对于如静态变量等非自动变量都会导 致编译器报错

4. lambda函数的应用

①用作简单的“局部函数”

C/C++语言标准中不允许局部函数（local function，或内嵌函数，nested function，即在函数作用域中定义的函数）的存在，但是lambda函数的使用可以实现这一点。相比于函数体外定义static inline函数，lambda函数没有实际运行时的绝对性能优势，但是代码可读性和作用域范围都和局部函数一致。

②值传递捕捉和引用传递捕捉

值传递捕捉的值在lambda函数定义的时候就已经决定，甚至说已经完成了值传递或拷贝。之后如果改变变量的值，并调用lambda函数，函数中的该变量仍然是改变之前的值。因此相当于该变量在传递之后成为了lambda函数的一个常量。

引用传递捕捉在任何时候lambda函数中的该变量都和父作用域中该变量的实际取值时时相同。

值传递捕捉const型（默认的）lambda函数不能改变值传递捕捉过来的常量的值，加上mutable修饰lambda函数之后可以修改。引用传递捕捉const型lambda函数则可以修改捕捉的参数。而对于lambda函数的参数（即正常意义下的函数参数）则遵循一般意义的函数参数规则，和捕捉参数无关。

# lvalues、rvalues、（左值）引用和右值引用

## 移动语义、完美转发

1. lvalues

左值，所有的变量，或者是常变量，可以取地址的，拥有名称的实体，都是lvalues

2. rvalues

右值，临时存在的数值，不能取地址，如数字表示的数值，临时的无名的对象等；rvalues 不能作为被赋值的量，即不能出现在“=”左边

•C++11中，rvalues包含两个概念，将亡值（xvalue，expiring value），另一个是和传统 右值含义一致的纯右值（prvalue），两个概念加上左值代表了全部c++11的数值。

•将亡值是c++11新增的跟右值引用相关的表达式，比如返回T&&类型的函数返回值， 这些函数返回的值确属右值，但是返回值不是拷贝构造的，是移动构造的。这相当于不 断的将一个本应该消失的右值不断续命。

3. 右值引用

c++98以前的版本，即传统意义上的引用都是定义在变量或者左值基础上的，所以又称 左值引用；c++11引入了对于右值的引用，也就是定义一个变量是一个右值的 别名。

左值引用：T & X = lvalue

右值引用：T && X = rvalue

•右值引用的意义和优势在于，定义的引用变量直接和右值（初始化值）关联，而普通 的定义方式是利用右值来构造被定义变量，之后再析构右值。因此对于尤其是使用自定 义类型的情况下，右值引用可以节约开销。

4. 几种引用方法比较

•c++98中，使用常量左值引用也是可以使用右值初始化的，但是各有利弊

T & X //只能接受非常量左值进行初始化，可写

const T & X //可以接受非常量左值，常量左值，以及右值进行初始化，但是不可写

T && X //只能接受非常量右值，可写

const T && X //只能接受右值，不可写，其功能被常量左值引用包含

5. 移动语义

c++11中，“偷走”临时变量中资源的构造函数，称为移动构造函数，比如使用右值（临时变量）来构造一个对象所使用的构造函数，构造后临时对象变为构造对象。对应的，另有拷贝构造函数用来使用已有对象建立新对象，两个对象共同存在。使用移动构造函数的行为和指令，称为移动语义（move semantics）。

一般的，移动构造函数在初始化参数为右值时用来代替拷贝构造函数节约性能开销，但是对于无法实现移动语义的情形，可同时声明一个常量左值引用为参数的拷贝构造函数（拷贝语义），可以应对各种情况。一般的，拷贝构造和移动构造函数都是同时声明的，保证类能够实现两种语义，因为自定义任何一个构造函数都会导致编译器不默认提供构造函数。

右值引用为参数的移动构造函数可以修改作为参数的临时对象。对于有指针成员的类型，如果使用常量左值引用为参数的构造函数，临时对象中的指针成员所指向的内存在构造函数结束后必然因为析构而释放，则必须为新对象的指针成员重新分配内存（是为拷贝）；而右值引用构造函数可以设定临时对象指针成员在为新对象指针成员赋值之后变为空值，析构后不影响新对象指针成员。

6. 判断模板中的引用类型

有的时候，我们可能不知道一个类型是否是引用类型，以及是左值引用还是右值引用（这 在模板中比较常见）。标准库在<type\_traits>头文件中提供了3个模板类：is\_rvalue\_reference、is\_lvalue\_reference、is\_reference，可供我们进行判断。

比如：cout<<is\_rvalue\_reference<string &&>::value;

我们通过模板类的成员value就可以打印出stirng&&是否是一个右值引用了。配合第4章中的类型推导操作符decltype，我们甚至还可以对变量的类型进行判断。当读者搞不清楚引用类型的时候，不妨使用这样的小工具实验 一下。

在<type\_traits>里，还有一些辅助模板类，用来判断类型是否是可以移动的（可以实现移动构造和移动赋值），比如is\_move\_constructible<X>、is\_trivially\_move\_constructible<X>、is\_nothrow\_move\_constructible<X>，仍然是考察成员value的值。

7. std::move和std::move\_if\_noexcept

标准库<utility>中提供了std::move(X)函数来将参数X强制转化为右值，用以实现移动语义；值得注意的是，右值引用和左值引用一样，引用的本质是一个有名字（别名）的变量，所以是左值。

move\_if\_noexcept函数在类的移动构造函数没有noexcept关键字修饰时，返回一个左值引用从而使变量可以使用拷贝语义，反之，正常返回右值引用，使用移动语义。这样做的目的是避免移动构造函数运行期间抛出异常，造成悬挂指针（dangling pointer）。

另外，std::move(X)的作用和static\_cast<T &&>(X)的作用是相同的。

8. 完美转发

①含义：指在函数模板中，完全按照模板的参数类型，将参数传递给函数模板中调用的另外 一个函数。通俗的讲，就是在一个函数A中调用另一个函数B，B使用的参数是A从 外界原样传递过来的，即A将参数转发给B。

②引用折叠（reference collapsing）：在模板类型的使用中，如果遇到多重引用的声明，遵循 “遇到左值引用优先”的原则将多重引用声明合并为简单的左值引用或右值引用

T & X //无论T是左值或右值引用，如int & & X或int && & X，最终合并为左值引用

T && X //T为左值引用时，最终为左值引用，T为右值引用时，最终为右值引用

•对模板类型参数推导时，如果采用T && X的参数声明方式，实参为左值时，T推导为T&， 实参为右值时，T推导为T&&，而不是T

③应用：主要应用于包装函数，即将多个功能类似参数大同小异的函数包装成一个函数模板， 函数模板用于转发参数给对应的函数。

template<typename T, typename U>

void PerfectForward(T &&t, U& Func) {

cout << t << "\tforwarded..." << endl;

Func( forward<T>(t));}

•forward是一个函数模板，用来完美转发，作用类似于static\_cast<T &&>(t)

# main()函数参数

通常我们在写主函数时都是void main()或int main() {..return 0;},但ANSI-C(美国国家标准协会,C的第一个标准ANSI发布)在C89/C99中main()函数主要形式为:

(1).int main(void)

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

其参数argc和argv用于运行时,把命令行参数传入主程序.其中ARG是指arguments,即参数.具体含义如下:

(参照Arguments to main和C++ Primer7.2.6节)

(1).int argc:英文名为arguments count(参数计数)

count of cmd line args,运行程序传送给main函数的命令行参数总个数,包括可执行程序名,其中当argc=1时表示只有一个程序名称,此时存储在argv[0]中.

(2).char \*\*argv:英文名为arguments value/vector(参数值)

pointer to table of cmd line args,字符串数组,用来存放指向字符串参数的指针数组,每个元素指向一个参数,空格分隔参数,其长度为argc.数组下标从0开始,argv[argc]=NULL.

argv[0] 指向程序运行时的全路径名

argv[1] 指向程序在DOS命令中执行程序名后的第一个字符串

argv[2] 指向执行程序名后的第二个字符串

argv[argc] 为NULL.

•另外int wmain()的写法表示用Unicode代替ASCII编码的程序，或者用\_tmain让编译器自动决定

# nothrow常量用法

std::nothrow一般只用作new和new []运算符的参数，用以调用不同的new重载函数。目的是当分配失败时禁止函数返回异常bad\_alloc，而是返回空指针。如：

new (std::nothrow) 类型;

# nullptr、NULL与指针空值

1. 指针空值（nullptr）

①int \* ptr = 0;

•C++98标准中，字面常量0既可以是一个整型，也可以是一个无类型的指针（void\*），一般情况下编译器总是优先将0作为整型常量处理

②int \* ptr = NULL;

•NULL是宏定义，在传统的C头文件（stddef.h）中，NULL可以是字面常量0，或者是无类型指针

③int \* ptr = nullptr;

•C++11新增内容，nullptr\_t为“指针空值类型”，nullptr为该类型的常量亦为关键字，且仅可以转化为任意指针类型（存在隐式转换），非指针类型无论如何不能转化：

<cstddef>头文件中定义如下，使用指针空值类型时必须include，使用nullptr不需

typedef decltype(nullptr) nullptr\_t;

2. 使用细则

①nullptr的隐式转换

nullptr的实际类型并非是指针类型，只是一定情况下允许隐式转换为指针类型（包括无类型指针类型），如对指针变量的赋值过程。在使用模板的情况下，编译器往往无法将nullptr推导为指针类型，因此需要手动显式类型转换。

②nullptr与(void\*)0的区别

nullptr和(void\*)0均可以显式转换为任意指针类型，但是只有nullptr存在隐式转换（C标准中后者也可以隐式转换为任意类型，C++不行），因此nullptr完全可以替代后者。

③nullptr的其他性质

nullptr被定义为一个右值常量，不能对nullptr进行取地址操作

# operator precedence & associativity运算符优先级与结合性

C 语言与其他高级语言相比, 一个显著的特点就是其运算符特别丰富, 共有34 种运算符。C 语言将这34 种运算符规定了不同的优先级别(precedence)和结合性(associativity)。优先级是用来标识运算符在表达式中的运算顺序的, 在求解表达式的值的时候, 总是先按运算符的优先次序由高到低进行操作, 可是, 当一个运算对象两侧的运算符优先级别相同时, 则按运算符的结合性来确定表达式的运算顺序。

运算符的结合性指同一优先级的运算符在表达式中操作的组织方向, C 语言规定了各种运算符的结合方向( 结合性) 。大多数运算符结合方向是“自左至右”, 即: 先左后右, 例如a- b+c, b 两侧有- 和+两种运算符的优先级相同, 按先左后右结合方向, b 先与减号结合, 执行a- b 的运算, 再执行加c 的运算。除了自左至右的结合性外, C 语言有三类运算符参与运算的结合方向是从右至左。即: 单目运算符, 条件运算符, 以及赋值运算符。

关于结合性的概念在其他高级语言中是没有的, 这是C语言的特点之一,特别是从右至左结合性容易出错。如当i=6 时, 求- i++的值和i 的值。由于“- ”(负号) “++”为同一个优先级, 故应理解为- (i++), 又因是后置加, 所以先有- i++的值为- 6, 然后i 增值1 为7, 即i=7。

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Operator** | **Description** | **Associativity** |
| **1** | :: | Scope resolution | Left-to-right |
| **2** | a++   a-- | Suffix/postfix [increment and decrement](https://en.cppreference.com/w/cpp/language/operator_incdec) |
| *type*()   *type*{} | [Functional cast](https://en.cppreference.com/w/cpp/language/explicit_cast) |
| a() | [Function call](https://en.cppreference.com/w/cpp/language/operator_other#Built-in_function_call_operator) |
| a[] | [Subscript](https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_subscript_operator) |
| .   -> | [Member access](https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_member_access_operators) |
| **3** | ++a   --a | Prefix [increment and decrement](https://en.cppreference.com/w/cpp/language/operator_incdec) | Right-to-left |
| +a   -a | Unary [plus and minus](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Unary_arithmetic_operators) |
| !   ~ | [Logical NOT](https://en.cppreference.com/w/cpp/language/operator_logical) and [bitwise NOT](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_logic_operators) |
| (*type*) | [C-style cast](https://en.cppreference.com/w/cpp/language/explicit_cast) |
| \*a | [Indirection](https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_indirection_operator) (dereference) |
| &a | [Address-of](https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_address-of_operator) |
| sizeof | [Size-of](https://en.cppreference.com/w/cpp/language/sizeof)[[note 1]](https://en.cppreference.com/w/cpp/language/operator_precedence#cite_note-1) |
| co\_await | await-expression (C++20) |
| new   new[] | [Dynamic memory allocation](https://en.cppreference.com/w/cpp/language/new) |
| delete   delete[] | [Dynamic memory deallocation](https://en.cppreference.com/w/cpp/language/delete) |
| **4** | .\*   ->\* | [Pointer-to-member](https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_pointer-to-member_access_operators) | Left-to-right |
| **5** | a\*b   a/b   a%b | [Multiplication, division, and remainder](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Multiplicative_operators) |
| **6** | a+b   a-b | [Addition and subtraction](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Additive_operators) |
| **7** | <<   >> | Bitwise [left shift and right shift](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_shift_operators) |
| **8** | <=> | [Three-way comparison operator](https://en.cppreference.com/w/cpp/language/operator_comparison#Three-way_comparison) (since C++20) |
| **9** | <   <= | For [relational operators](https://en.cppreference.com/w/cpp/language/operator_comparison) < and ≤ respectively |
| >   >= | For [relational operators](https://en.cppreference.com/w/cpp/language/operator_comparison) > and ≥ respectively |
| **10** | ==   != | For [relational operators](https://en.cppreference.com/w/cpp/language/operator_comparison) = and ≠ respectively |
| **11** | & | [Bitwise AND](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_logic_operators) |
| **12** | ^ | [Bitwise XOR](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_logic_operators) (exclusive or) |
| **13** | | | [Bitwise OR](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_logic_operators) (inclusive or) |
| **14** | && | [Logical AND](https://en.cppreference.com/w/cpp/language/operator_logical) |
| **15** | || | [Logical OR](https://en.cppreference.com/w/cpp/language/operator_logical) |
| **16** | a?b:c | [Ternary conditional](https://en.cppreference.com/w/cpp/language/operator_other#Conditional_operator)[[note 2]](https://en.cppreference.com/w/cpp/language/operator_precedence#cite_note-2) | Right-to-left |
| throw | [throw operator](https://en.cppreference.com/w/cpp/language/throw) |
| co\_yield | yield-expression (C++20) |
| = | [Direct assignment](https://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_direct_assignment) (provided by default for C++ classes) |
| +=   -= | [Compound assignment](https://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_compound_assignment) by sum and difference |
| \*=   /=   %= | [Compound assignment](https://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_compound_assignment) by product, quotient, and remainder |
| <<=   >>= | [Compound assignment](https://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_compound_assignment) by bitwise left shift and right shift |
| &=   ^=   |= | [Compound assignment](https://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_compound_assignment) by bitwise AND, XOR, and OR |
| **17** | , | [Comma](https://en.cppreference.com/w/cpp/language/operator_other#Built-in_comma_operator) | Left-to-right |

1. ↑ The operand of sizeof can't be a C-style type cast: the expression sizeof (int) \* p is unambiguously interpreted as (sizeof(int)) \* p, but not sizeof((int)\*p).

2. ↑ The expression in the middle of the conditional operator (between **?** and **:**) is parsed as if parenthesized: its precedence relative to ?: is ignored.

# POD类型

1. POD

Plain Old Data，是一种类型的属性，尤其是用户自定义类型的属性，比如类和结构体。表达了一种“普通”，“与C兼容”的属性。具体划分为两个基本概念的集合。

2. 平凡性（trivial）

POD的一个基本概念，通常一个平凡的类或结构体应该符合：

①拥有平凡的默认构造函数和析构函数（trivial constructor/destructor）

即用户不定义任何构造析构函数（即便是空函数），由编译器默认生成的；可以通过=default来显式声明缺省版本的构造函数

②拥有平凡的拷贝构造函数和移动构造函数（trivial copy/move constructor）

③拥有平凡的拷贝赋值运算符和移动赋值运算符（trivial assignment/move operator）

④不能包含虚函数以及虚基类

•使用<type\_traits>的is\_trivial<X>::value可以判断X类型的平凡性，平凡为1，不平凡为0

3. 标准布局性（standard layout）

①所有非静态成员具有相同的访问权限

即所有非静态成员访问属性相同，是public或private或protected之一

②类或结构体继承时，满足以下两种情况之一，则派生类为标准布局

•派生类中有非静态成员，且只有一个仅包含静态成员的基类

•基类有非静态成员，而派生类没有非静态成员

注：多重继承时，一旦非静态成员出现在多个基类中，派生类也不属于标准布局

③类中第一个非静态成员的类型与其基类不同

这种情况下，如果基类为空，则派生类第一个成员和基类共享地址，否则因为类型一致，必须给基类留出一个字节空间

④没有虚函数和虚基类

⑤所有非静态数据成员均符合标准布局类型，其基类也符合标准布局

•使用<type\_traits>的is\_standard\_layout<X>::value可以判断X类型的标准布局性

•使用<type\_traits>的is\_pod<X>::value可以判断X类型是否为POD

4. POD用途

①字节赋值，代码中可以安全的使用memset和memcpy对POD类型进行初始化和拷贝等操作

②提供对C内存布局兼容，POD类型在C和C++之间操作总是安全的

③保证了静态初始化的安全有效，POD类型的初始化更加简单

# preprocessing预处理

## preprocessor macro预定义宏和预定义标识符（C99，C++11兼容）

预定义宏和预定义标识符名称通常由双下划线\_\_开头与结尾\_\_。

1.预定义宏

预定义宏多数都是反映编译器所针对的机器平台信息或某些库的支持性的。因此对于多目标平台编程有重要的意义。

·标准宏

|  |  |
| --- | --- |
| \_\_FILE\_\_ | 表示当前文件（路径）名称的字符串 |
| \_\_LINE\_\_ | 表示当前文件行位置 |
| \_\_DATE\_\_ | 表示程序执行日期 |
| \_\_TIME\_\_ | 表示程序执行的时间时刻 |
| \_\_STDC\_HOSTED\_\_ | 1表示编译器目标系统环境包含完整的标准C库 |
| \_\_STDC\_\_ | 表示编译器的实现是否和C标准一致 |
| \_\_STDC\_VERSION\_\_ | 表示所支持的C标准的版本，格式为yyyymmL |
| \_\_STDC\_ISO\_10646\_\_ | 表示编译环境符合某个版本的ISO/IEC 10646标准 |
| \_\_cplusplus | 表示当前使用的是C++编译器 |
| \_\_OBJC\_\_ | 1表示编译器支持objective-C |
| \_\_ASSEMBLER\_\_ | 1表示预处理汇编语言 |

2.预定义标识符

\_\_func\_\_

表示所在函数（C99）或结构体/类（C++11）的名称字符串，实际上由编译器在函数体中初始位置定义

static const char\* \_\_func\_\_ = “xxx”;

也因此在函数头中是无法使用\_\_func\_\_的，因为还未定义

## preprocessor directives预处理命令和\_Pragma操作符

1.预处理命令

#include

包含其他/头文件

#ifdef, #ifndef, #if, #endif, #else, #elif

条件编译

#define, #undef

宏定义

#pragma once

只编译一次

#pragma region XXX

#pragma endregion

定义代码块，XXX为代码块注释信息

2.\_Pragma操作符

\_Pragma操作符的用法和sizeof是类似的，于C++11中引入。格式为

\_Pragma (字符串字面量) // 如\_Pragma (“once”)

其功能和#pragma相同，但是由于不是预处理命令，使用范围比#pragma更广。

# printf与格式化输出

printf和格式化输出的概念在C/C++和Java等各种语言环境下都存在。其通常的原型是

printf ( const char \* format, ... );

首个参数是个用于定义格式的字符串，也可以说是一个正则表达式。后续的变长参数用于传递在字符串中依次嵌入的变量值。正则表达式遵循以下格式：

%[flags][width][.precision][length]specifier

1. specifier

最主要且必备的元素：

|  |  |  |
| --- | --- | --- |
| ***specifier*** | **Output** | **Example** |
| d *or* i | Signed decimal integer | 392 |
| u | Unsigned decimal integer | 7235 |
| o | Unsigned octal | 610 |
| x | Unsigned hexadecimal integer | 7fa |
| X | Unsigned hexadecimal integer (uppercase) | 7FA |
| f | Decimal floating point, lowercase | 392.65 |
| F | Decimal floating point, uppercase | 392.65 |
| e | Scientific notation (mantissa/exponent), lowercase | 3.9265e+2 |
| E | Scientific notation (mantissa/exponent), uppercase | 3.9265E+2 |
| g | Use the shortest representation: %e or %f | 392.65 |
| G | Use the shortest representation: %E or %F | 392.65 |
| a | Hexadecimal floating point, lowercase | -0xc.90fep-2 |
| A | Hexadecimal floating point, uppercase | -0XC.90FEP-2 |
| c | Character | a |
| s | String of characters | sample |
| p | Pointer address | b8000000 |
| n | Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location. |  |
| % | A % followed by another % character will write a single % to the stream. | % |

2. flags

|  |  |
| --- | --- |
| ***flags*** | **description** |
| - | Left-justify within the given field width; Right justification is the default (see *width* sub-specifier). |
| + | Forces to preceed the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign. |
| *(space)* | If no sign is going to be written, a blank space is inserted before the value. |
| # | Used with o, x or X specifiers the value is preceeded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written. |
| 0 | Left-pads the number with zeroes (0) instead of spaces when padding is specified (see *width* sub-specifier). |

3. width

|  |  |
| --- | --- |
| ***width*** | **description** |
| *(number)* | Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger. |
| \* | The *width* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted. |

4. precision

|  |  |
| --- | --- |
| ***.precision*** | **description** |
| .*number* | For integer specifiers (d, i, o, u, x, X): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of 0 means that no character is written for the value 0. For a, A, e, E, f and F specifiers: this is the number of digits to be printed **after** the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for *precision*, 0 is assumed. |
| .\* | The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted. |

5. length

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **specifiers** | | | | | | |
| ***length*** | **d i** | **u o x X** | **f F e E g G a A** | **c** | **s** | **p** | **n** |
| *(none)* | int | unsigned int | double | int | char\* | void\* | int\* |
| hh | signed char | unsigned char |  |  |  |  | signed char\* |
| h | short int | unsigned short int |  |  |  |  | short int\* |
| l | long int | unsigned long int |  | [wint\_t](http://www.cplusplus.com/wint_t) | wchar\_t\* |  | long int\* |
| ll | long long int | unsigned long long int |  |  |  |  | long long int\* |
| j | intmax\_t | [uintmax\_t](http://www.cplusplus.com/uintmax_t) |  |  |  |  | [intmax\_t](http://www.cplusplus.com/intmax_t)\* |
| z | [size\_t](http://www.cplusplus.com/size_t) | [size\_t](http://www.cplusplus.com/size_t) |  |  |  |  | [size\_t](http://www.cplusplus.com/size_t)\* |
| t | [ptrdiff\_t](http://www.cplusplus.com/ptrdiff_t) | [ptrdiff\_t](http://www.cplusplus.com/ptrdiff_t) |  |  |  |  | [ptrdiff\_t](http://www.cplusplus.com/ptrdiff_t)\* |
| L |  |  | long double |  |  |  |  |

Note regarding the c specifier: it takes an int (or [wint\_t](http://www.cplusplus.com/wint_t)) as argument, but performs the proper conversion to a char value (or a wchar\_t) before formatting it for output.

# RNG随机数生成器

随机数生成的是伪随机数，通过一个特殊的递推关系决定

有关指令和更详细的内容参见“C++常用函数和语句整理”

# smart pointer智能指针

由c++11引入，可以解决以下问题：

dangling pointer——指针的指向内容已释放，但指针仍被使用

memory leaks——分配的内存未被释放，但已经无用

double deletion——两个不同的指针释放同一内存区域

·所有的智能指针本质上都是管理动态分配（new运算符所创建）的对象。

1. unique pointer(unique\_ptr)

需要include <memory>

唯一指针指向的对象具有唯一的归属性，如果该指针脱离作用域或改为指向其他唯一对象，则原对象被自动释放。唯一指针也可以将所有权转移给其他唯一指针。

①声明

unique\_ptr<int> uptr(new int);

unique\_ptr<int[]> uptr(new int[5]);

unique\_ptr<int> uptr = make\_unique<int>();

unique\_ptr<int[]> uptr = make\_unique<int[]>(5);

②转移所有权

uptr2 = move(uptr);

uptr.reset(new int); // 转移uptr的指向至新对象

③手动释放内存

uptr = nullptr; 或 uptr.reset();

2. shared pointer(shared\_ptr)

共享指针可以指向相同的动态分配对象，并且有一个control block来管理指针的指向情况，实际就是监督动态分配对象的引用数目以释放引用数为0的对象。

①声明

shared\_ptr<int> sptr(rawPtr); // 由原始指针构造

shared\_ptr<int> sptr2 = sptr; // 赋值构造

shared\_ptr<int> sptr = make\_shared<int>(); // 原生构造（即不从原始指针构造）

·make\_shared<T>(X)使用X作为T的构造参数，以new运算符分配空间，返回该空间的共享指针。

·注意对于同一个动态分配对象，每调用一次原始指针构造函数，都会制造一个control block，且彼此不会共享使用次数。若其中一个指针销毁，其指向对象便会被视为零引用而析构，因此还是会产生悬挂指针的问题。故其余共享指针只能用赋值方式初始化。

②查看引用次数

long int use\_cout() const noexcept

③强制类型转换

shared\_ptr有对应的强制类型转换函数模板，包括static\_pointer\_cast，dynamic\_pointer\_cast，和const\_pointer\_cast。分别对应类似名称的非智能指针版本，用法类似。

3. weak pointer(weak\_ptr)

weak\_ptr可以指向shared\_ptr指针指向的对象内存，但是并不计入引用计数内。使用weak\_ptr成员lock()可以返回指向该内存的一个shared\_ptr对象，若对象内存已经无效，则返回nullptr。可用于验证shared\_ptr的有效性。

·更多有关智能指针的用法需参考其他详细内容。

# static关键字的用法和含义以及global全局变量

static的含义和用法是非常混乱的（因为这个词在计算机行业被用烂了），首先static duration的概念是指静态作用域，即一个标识符拥有从程序至始至终的生命期，对应的automatic duration是自动作用域，即限于当前代码块的生命期。

1.修饰全局变量

这是最最容易混淆的static作用，它使得全局变量仅作用于当前文件范围。

2.修饰局部变量

局部静态变量local static在作用层面和全局变量global是一样的，且二者均存储于虚拟内存的固定位置。但是除了明显地声明的位置不同，还存在一个可能的重要差别（视编译器对C++标准的具体实现而定）——即全局变量是在主函数运行前就完成初始化的，局部静态变量是在控制流第一次抵达其初始化代码位置才执行的。这里面还有很多复杂的问题值得讨论，见后面几个小节的分类讨论。

3.全局变量的初始化顺序问题

在同一个文件中，全局变量的初始化一般按照代码顺序进行。但分别定义在不同文件中的全局变量就可能产生不确定的初始化顺序，如果它们之间有依赖关系，则可能造成运行时错误。这也是全局变量不被鼓励使用的主要原因之一。

4.静态局部变量的初始化时机

对于静态局部变量，编译器一般选择Construct On First Use Idiom，即使用时再构造的策略。其主要目的之一就是避免与全局变量之间出现上述初始化顺序的问题，另外也可以提高程序运行前的“准备”速度。

# stream manipulator流操作符改变cin/cout格式

需要<iomanip>头文件，通过在cin/cout流运算符>>/<<之后添加流操作符stream manipulators来改变输入输出格式和其他相关属性。

1. 常用cin流操作符

setw(): 设定输入的最大宽度，可用于输入C-String

·cin.width(SIZE) 也可以起到这个限制

2. 常用cout流操作符

（1）只影响下一个显示值

setw(x): 设定输出的宽度至少为x个空间（space wide）

·若数据本身多于x个空间宽，最终输出宽度等于数据的真实宽度，否则在左边补齐至x宽度（即右对齐right-justified）。补位字符不是0，而是空格，这对于数值类型数据的输出也是如此，亦即这不是数值精度的设定。

·小数点、空格均占空间

（2）影响至改变设定为止

fixed: 用decimal notaion而不是E-notation显示浮点数

setprecision(x):

·有fixed时，x为小数点decimal后位数

·无fixed时，x为有效位数significant digits

·采用四舍五入

showpoint: 永远显示小数点，与fixed或setprecision合用

noshowpoint: 不显示小数点，对于小数点后有非零数字的情况无效，但是会覆盖precision对小数点后零的个数的设定，亦即绝不会出现类似30.00的数（而是30）

left: 左对齐，右补位

right: 类似上

# struct结构体

结构体中只含有数据成员，而且数据成员按照顺序存储，也就是说结构体数组和其所有数据成员按顺序组成的数组是一样的存储方式和结构。在拷贝到缓存中之后，二者之间没有差别。

# template模板

1.基本用法

具体参考基础知识和各种网络资源即可

2.模板参数

能够作为模板的参数有两种：一种是class或typename修饰的类型参数，一种是具体类型的数据参数（就像一般函数参数一样）。然而具体类型的数据参数只能是非浮点类型的。

3.特化specialization

特化即为对模板的某些实例进行专门的定义，以和一般的模板内容相区别。特化分为全特化和偏特化。

全特化是指将所有模板参数完全指定的实例化，或就是某种具体实现。偏特化是指对原模板参数进行各种形式地进一步限制而定义的新模板，注意偏特化不仅仅是部分完全指定模板参数的行为。举例而言，对于class T模板参数特化为int时，视为全特化；特化为T\*或T&时或有两个模板参数并指定一个为int时，视为偏特化。

在定义时，模板特化必须在初始模板primary template定义后进行。初始模板的名称标识符后不加<模板参数列表>，而模板特化则需要添加以指定特化内容。注意，模板特化不能和初始模板冲突，且不当的特化和调用有可能产生歧义。

4.显式实例化

一般而言，C++的模板的声明和定义都会放于同一个头文件中，否则会导致编译器报错。如果强行要进行声明与实现分离，可以使用显式实例化。按照一般方式在头文件声明，源文件定义后，在定义所在源文件的末尾进行实例化：

template class 模板名称<实例化类型参数>;

这种方式显然限制了程序其他位置所能使用的模板实例化种类，即这样定义的模板只有显式实例化的那几种可用。

## typename关键字的用法、name lookup名称解析

1.名称解析

①dependent names和non-dependent names

dependent是指和模板参数有关，或必须由模板参数确定的标识符。具体的定义详参文档。

②qualified name和unqualified name

qualified name是指处于作用域限制符::右边的标识符，反之unqualified是指左侧不加::的标识符。两种name遵循不同的lookup解析准则。

③Argument-dependent lookup, ADL

主要用于带参数实体名称的解析，如函数。

例如std::cout << “xxx”; 等效于operator<<(std::cout, “xxx”); 但是operator<<只有在std空间中才重载了接受std::cout参数的形式。由于其参数处于std空间，因而编译器会在std空间中解析<<，因此不加std::operator<<也可以。

④Elaborated type specifier

同一个结构体、类或命名空间内的两个类型和变量若同名，在使用qualified name形式访问时，如A::X，则编译器会自动将对该名称的访问解析为变量名，它不会因为该变量是非静态的或私有的而转而解析为类型，进而只能报错（除非不重名或在访问时再次使用显式的class/struct/union关键字指明，此时它们的作用是Elaborated type specifier）。

2.typename用法

先引用和翻译cppreference给的解释：

①In a template declaration, typename can be used as an alternative to class to declare type template parameters and template template parameters (since C++17). 定义模板时，声明模板类型参数的关键字，可以和class互换。

②Inside a declaration or a definition of a template, typename can be used to declare that a dependent qualified name is a type. 在模板定义中，可以用来标明一个dependent qualified name是个类型，而非变量。若不加则会被编译器认为是一个变量名。

·模板定义非常灵活且容易引发大量歧义，如

int x;

template <class T>

struct A

{

T::X\* x;

}

根据模板参数T的不同X可以解析为类型，进而便是指针声明；也可以解析为静态变量X和全局变量x的乘法。因此所有模板中的依赖于模板参数的dependent、含有作用域限制符::的qualified的标识符，均需要消歧义。

·另一个例子是

template <class T>

struct A

{

B<T>::X\* x;

}

虽然看似B中的X若为变量且可能类型会变化，但不影响X是类型还是变量。然而由于有特化specialization的存在，B中的X完全可以在不同的实例化中声明为类型和变量。

·上述二例都是在正常使用作用域限制符的情况下能够产生歧义的例子。实际上由于模板工作方式的特殊性（不实例化调用的时候不进行编译），是可以并且必须使用作用域限制符修饰未在当前模板块中声明的变量的（哪怕是继承的或非静态变量）。

· typename关键字虽然提示编译器在处理某个dependent qualified names，如T::X时，认为其是类型，但若在该作用域内确实存在同名的变量和类型时，并不能起到Elaborated type specifier的区分作用。这和typename并没有改变其修饰的标识符的名称解析规则有关。进一步而言Elaborated type specifier，尤其是class是可以代替typename的第1、2项功能的，且兼具改变名称解析规则的功能。

③Inside a declaration or a definition of a template, (until C++11) typename can be used before a non-dependent qualified type name. It has no effect in this case. 现在可以忽略的功能

④Inside a requirements for type requirements (since C++20)

## variadic function变长函数、变长宏和变长模板

1.变长参数的宏定义（源于C）

指在宏定义中参数列表的最后一个参数为省略号，而预定义宏\_\_VA\_ARGS\_\_则可以在宏定义的实现部分替换省略号所代表的字符串。如：

#define PR(…) printf (\_\_VA\_ARGS\_\_)

·其实就是在宏的层面支持变长参数函数

2.变长函数（源于C）

需要借助stdarg.h/<cstdarg>头文件。

·变长函数的声明使用…省略号表示更多参数，注意最后一个具名参数后仍有逗号“，”。且更多省略参数也不指定类型。

·在函数体中使用va\_list类型声明用于存储参数列表的变量。

·然后使用va\_start (va\_list ap, paramN)宏函数使ap获得参数paramN之后的参数列表。paramN只是一个形参的名称。

·再使用va\_arg (va\_list ap, type)宏函数返回声明为type类型的参数，每次调用顺序性地返回一个参数，且由程序员自行负责参数实际类型和type的匹配。另外，va\_arg也不负责对参数总个数的检查，所以一般都会将不定参数总数作为一个具名参数（尤其是最后一个具名参数）声明出来。

·最后在函数体中用va\_end (va\_list ap)在函数返回前完成清理工作。

值得一提的是，C语言中printf函数就是变长函数，它也正是利用格式化符号来判断接受的参数列表的信息的。

3.变长模板类（C++11）

1）声明方法

template <typename… T> class A

// 即引入了新的模板参数类型，称为template parameter pack。和一般的模板参数用法类似，只是一次性推导为若干个类型而已。另外，typename也可以是具体类型，即非类型模板参数。

2）使用方法

①传递给其他模板

B<T…>

// 这被称为包扩展pack expansion表达式，即将模板参数包T解包传递给B模板

②实例化

如一般模板正常实例化即可

③偏特化递归定义模板

由于模板参数包是一个整体，且用模板常规的使用方式无法随意改变模板类的成员个数。除非使用模板特化和递归定义/继承的方式实现，例如STL::tuple类就是偏特化递归继承实现的动态个数成员。

template <typename… Elements> class tuple;

template <typename Head, typename… Tail>

class tuple <Head, Tail…> : private tuple <Tail…>

{

Head head;

};

template <> class tuple <> {};

·另外这种方式结合非类型模板参数还可用于模板元编程。

4.变长模板函数

声明（例子）：

template <typename… T> void f (T… args);

除了模板参数包T，模板函数还有函数参数包args（一般往往由T解包声明），且必须是函数最后一个参数。模板参数包和函数参数包都可以解包。模板函数也可以递归调用将变长参数进行逐一分解处理，不过不需要特化定义，但需要定义递归基版本。

5.包扩展

除了直接解包展开参数，还可以实现不同的解包方式，如：

template <typename… A>

A&&… // 可以解包为右值引用序列

B<A>…// 可以实现解包后分别赋值，等效于B<A1>, B<A2>……

void f(A… a)

{

s(g(a)…); //类似上例，注意这种逗号展开形式也只能作为函数参数

t(h<A>(a)…); //模板参数和函数参数双包齐开

}

·除此之外，还引入了sizeof…()操作符，用以计算参数包中的参数个数。因此sizeof…操作符可用于在某些情况下实现非递归的变长模板。

# typedef用法

typedef用于给类型名称起一个别名，有以下两个作用：

①一般用于将一个实际类型赋予一个符合程序逻辑且可增强程序可读性的名称，如typedef unsinged int studentID，即将无符号整型赋予studentID的别名，因此使用studentID声明变量即可起到同样实际作用；

②另外别名还可以起到简化书写的作用，比如函数指针的类型名称往往较长（因为包括参数列表），使用别名可简化成如同int a方式的使用方法。

·用法：

typedef的用法可以简单理解为：先声明一个该实际类型的变量，然后前加typedef即可将该变量名声明为类型别名。其实就是函数指针这种类型的声明比较特殊，大部分都遵循——

typedef 类型（可包括自定义类型的轻量级实现） 别名

的方式。

# 最小垃圾回收

1.安全派生safely derived指针

安全派生指针是指向由new分配的对象或其子对象的指针，且仅限于以下操作：

①在解引用基础上的引用

②定义明确的指针操作

③定义明确的指针转换

④指针和整型之间的转换

2.最小垃圾回收标准

现在只是C++11提出的标准，是否被编译器实现还不一定。由于C++中指针使用特别灵活，想要实现较简单且全面的垃圾回收机制非常困难，因此提出安全派生指针概念，即最小垃圾回收将只针对这些安全派生指针运作，至于“不安全”的指针需要程序员提醒垃圾回收器或自行处理。以下是最小垃圾回收标准的接口：

①确认最小垃圾回收特性的支持

pointer\_safety get\_pointer\_safety() noexcept

返回：

pointer\_safety::strict 支持全部概念

pointer\_safety::preferred 垃圾回收器只负责内存泄漏检测

pointer\_safety::relax 不支持

②声明非安全派生指针作用域——可达声明

void declare\_reachable(void\* p); //声明作用域开始

template <class T> T \*undeclare\_reachable(T \*p) noexcept //声明作用域结束

③声明连续内存区域的可达性

void declare\_no\_pointers(char \*p, size\_t n) noexcept

void undeclare\_no\_pointers(char \*p, size\_t n) noexcept