# 2017/10/12 Thu

## 第二章 C++概览：基础知识

1. complex复数类
2. 使用花括号内的一组初始化列表(p35)

double d{2.3};

complex<double> z2{d, d};

complex<double> z3 = {1,2};//当使用{…}的时候，符号=是可选的。

注意：使用初始化器列表的形式，不能出现信息丢失的类型转换(如窄化类型转换)

int i2{7.2};//错误，试图执行浮点数向整数的类型转换

1. constexpr：大致意思是“在编译时求值”。主要用于说明常量，作用是允许数据置于只读内存中（不可能被破坏）以及提升性能。(p37)

const int dmv = 17;//dmv is a const

int var = 17;//var is not a const

constexpr double max1 = 1.4\*square(dmv);//如果square(17)是常量表达式，则正确

constexpr double max2 = 1.4\* square(dmv);//错误：var不是常量表达式

注意：如果某个函数用在常量表达式( constant expression )中，及该表达式在编译时求值，则该函数必须定义成constexpr。如：

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

要想定义成constexpr，函数必须非常简单：函数中只能有一条用于计算某个值的return语句。constexpr函数可以接受非常量实参，但此时其结果将不会是常量表达式。当程序的上下文不需要常量表达式时，我们可以使用非常量表达式实参来调用constexpr函数，这样我们就不用吧同一个函数定义两次了。

1. 范围for语句(p39)

void print()

{

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

for ( auto x : v )//对v中的每个x

cout<<x<<’\n’;

for ( auto x : { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 } )

cout<<x<<’\n’;

}

如果我们不希望把v中的值拷贝到变量x中，而只想领x指向一个元素，则可如下书写代码：

void increment()

{

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

for ( auto& x : v )//引用

++x;

}

1. 空指针用nullptr，用nullptr能避免在整数（如0或NULL）和指针之间发生混淆

# 2017/10/15 sun

1. 枚举（p43）

强类型枚举，enum后面的class指明了枚举是强类型的

enum class Color { red, blue, green };

enum class Traffic\_light { green, yellow, red };

Color col = Color::red;

Traffic\_light light = Traffic\_light::red;

枚举类型常用于描述规模较小的整数值集合。通过使用有指代意义（且易于记忆的）举值名字可提高代码的可读性，降低出错的风险。

Color x = red;//错误：哪个red？

Color y = Traffic\_light::red;//错误：这个red不是Color的对象

同样，我们也不能隐式地混用Color和整数值：

int i = Color::red;//错误：Color::red不是一个int

Color c = 2;//错误：2不是一个Color对象

如果你不想显示第限定枚举值名字，并且希望枚举值可以是int（无需显示转换），则应该去掉enmu class中的class而得到一个“普通”的enum。

枚举类型是一种用户自定义类型，那么我们可以为他定义别的运算符：

Traffic\_light& operator++( Traffic\_light& t)//前置递增运算符

{

switch(t)

{

case Traffic\_light::green:return t = Traffic\_light::yellow;

case Traffic\_light::yellow: return t = Traffic\_light::red;

case Traffic\_light::red:return t = Traffic\_light:: green;

}

}

Traffic\_ligh next = ++light;//next编程了Traffic\_light::green

1. 静态断言 编译时进行检查(p49)

static\_assert(4<=sizeof(int)), “integers is too small”);//检查整数的尺寸

我们把这种表达某种期望的语句成为断言(assertion)

static\_assert机制能用于任何可以表达为常量表达式的东西

constexpr double C = 299792.458;

void f(double speed)

{

const double local\_max = 160.0/(60\*60);

static\_assert(speed<C, “can’t go that fast”);//错误，速度必须是个常量

static\_assert(local\_max<C, “can’t go that fast”); //OK

}

## 第三章 C++概览：抽象机制

1. 初始化容器（p55）

用于定义初始化器列表构造函数的std::initializer\_list是一种标准库类型，编译器可以辨识它：当我们使用{}列表时，如{1,2,3,4}，编译器会创建一个:initializer\_list类型的对象并将其提供给程序，因此我们可以书写：

Vector v1 = { 1, 2, 3, 4, 5 };//v1包含5个元素

Vector的初始化器列表构造函数可以定义成如下的形式：

Vector::Vector( std::initializer\_list<double> lst )//用一个列表初始化

:elem{new double[lst.size()]}

,sz{lst.size()}

{

copy( lst.begin(), lst.end(), elem );

}

1. 移动容器（p64）

class Vector

{

Vector(const Vector& a);//拷贝构造函数

Vector& operator=(const Vector& a);//拷贝赋值运算符

//不接受const实参,毕竟移动构造和赋值 最重要删除它实参中的值

Vector(Vector&& a);//移动构造函数

Vector& operator=(Vector&& a);//移动赋值运算符

}

Vector::Vector(Vector&& a)

:elem{a.elem}

,sz{a.sz}

{

a.elem = nullptr;

a.sz = 0;

}

符号&&的意思是“右值引用”，我们可以给该引用绑定一个右值。

左值 的大致含义是“能够出现在赋值运算符左侧的内容”

右值 的大致含义是“我们无法为其赋值”

右值引用 的大致含义是“引用了一个别人无法赋值的内容”

程序员可以知道一个值在什么地方不再被使用，但是编译器做不到这一点，因此程序员最好在程序中写的明确一些：

Vector f()

{

Vector x(1000);

Vector y(1000);

Vector z(1000);

//…

z = x;//执行拷贝操作

y = std::move(x);//执行移动操作

//…

return z;//执行移动操作

}

标准库函数move()负责返回实参的右值引用。

1. 抑制操作

对于层次中的类来说，使用默认的拷贝或者移动操作常常意味着风险：因为只给出一个基类的指针，我们无法了解派生类有什么样的成员，当然也不知道如何操作他们。因此最好的做法就是删除掉默认的拷贝和移动操作。

class Shape

{

public:

Shape(const Shape&) = delete;//没有拷贝操作

Shape& operator=(const Shape&) = delete;

Shape(Shape&&) = delete;//没有移动操作

Shape& operator=(Shape&&) = delete;

~Shape();

//…

}

如果使用者在类中显示地声明了析构函数，则移动操作将不会隐式地生成。而在本例中拷贝操作的生成也被禁止了。这就是为什么即使编译器可以隐式地提供析构函数，也最好显示地自己定义一个析构函数的原因。

这种 =delete的机制时通用的，也就是说，我们可以用它抑制任何操作。

1. 模板是一种编译时的机制，因此与“手动编写的代码”相比，并不会产生任何额外的运行时开销。
2. std:: accumulate ()

MSDN:Computes the sum of all the elements in a specified range including some initial value by computing successive partial sums or computes the result of successive partial results similarly obtained from using a specified binary operation other than the sum.

1. lambda表达式（p70）

void f(const Vector<int>& vec, const list<string>&lst, int x, const string& s)

{

cout<<”number of values less then ”<<x

<<”: ”<<count(vec, [&](int a){ return a<x; } )

<<’\n’;

cout<<”number of values less then ”<<s

<<”: ”<<count(lst, [&](const string& a){ return a<s; } )

<<’\n’;

}

这里的[&](int a){ return a<x; }被称为lambda表达式（lambda exprssion），他生成一个函数对象。[&]是一个捕获列表（capture list），它指明所用的局部名字（如x）将通过引用访问。如果我们希望只“捕获”x，则可以写成[&x]; 如果希望给生成的函数对象传递一个x的拷贝，则写成[=x]。什么也不捕获是[]，捕获所有通过应用访问的局部名字是[&]，捕获所有以值访问的局部名字是[=]。

1. 可变参数模板

template<typename T, typename… Tail>

void f(T head, Tail… tail)

{

g(head);//对head做某些操作

f(tail…);//再次处理tail

}

void f(){};//不执行任何操作

template<typename T>

void g(T x)

{

cout<<x<< “ ”;

}

实现可变参模板的关键是：当你传给它多个参数时，谨记把第一个参数和其他参数区别对待。此处，我们首次处理第一个参数(head)，然后使用剩余参数(tail)递归地调用f()。省略号…表示列表的“剩余部分”。最终tail将变成空，我们需要一个独立的函数来处理它。

1. 别名//感觉很像typedef，但是用法更加友好

如：using size\_t = unsigned int;

template<typename T>

class Vector

{

public:

using value\_type = T;

//…

};

template<typename Key, typename Value>

class Map

{

//…

};

template<typename Value>

using String\_map = Map<string, Value>;

String\_map<int> m;//m是一个Map<string, int>

1. 输入（p79）

void hello()

{

cout<<”please enter your name\n”;

string str;

cin>>str;

cout<<”Hello, ”<<str<<”!\n”;

}

如果键入Eric,程序将回应：

Hello,Eric!

默认情况下，空格等空白符会终止输入。因此，如果你键入Eric Bloodaxe冒充不幸的约克王，程序回应仍会是：

Hello,Eric!

你可以用函数getline()来读取一整行（包括结束时的换行符），例如：

void hello\_line()

{

cout<<”please enter your name\n”;

string str;

getline(cin, str);

cout<<”Hello, ”<<str<<”!\n”;

}

运行这个程序，再输入Eric Bloodaxe就会得到想要的输出：

Hello, Eric Bloodaxe!

行尾的换行符被自动丢弃掉了，因此接下来在cin的话会从下一行开始。

char c;

cin is;

is>>c;//默认跳过空格

is.get(c);//不跳过空格

# 2017/10/16 son

## 第四章 C++概览：容器与算法

1. 流迭代器（p91）

为了创建一个ostream\_iterator，我们需要指出使用哪一个流，以及输出的对象类型。如：

ostream\_iterator<string> oo{cout};

向\*oo赋值的作用是把所赋的值输出到cout。例如：

int main()

{

\*oo = “Hello,”;//等价于cout<<”Hello,”

++oo;//类似于用指针想数组中写入值

\*oo = “world!\n”;//等价于cout<<”world!\n”

}

## 第五章 C++概览：并发与实用功能

1. unique\_ptr与shared\_ptr（p97）

unique\_ptr 对应所有权唯一的情况

shared\_ptr 对应所有权共享的情况

shared\_ptr与unique\_ptr很多方面都相同，唯一却别是shared\_ptr对象使用拷贝操作，而非移动操作。某个对象的shared\_ptr共享该对象的所有权，只有当最后一个shared\_ptr被销毁时才被销毁。

1. 线程

void f();//函数

struct F//函数对象

{

void operator()();//F调用运算符

}

void user()

{

thread t1{f};//f()在独立的线程中执行

thread t2{F()};//F()()在独立的线程中执行

t1.join();//等待t1完成

t2.join();//等待t2完成

}

1. 任务参数

void f(vector<double>& v);//处理v的函数

struct F

{

vector<double>& v;

F(vector<double>& vv):v{vv}{}

void operator()();//调用运算符

}

int main()

{

vector<double> some\_vec{1,2,3,4,5,6};

vector<double> vec2{10,11,12,13,14};

thread t1{f, some\_vec};//f(some\_vec)在一个独立线程中执行

thread t2{F{vec2}};//F(vec2)在一个独立的线程中执行

t1.join();

t2.join();

}

1. 共享数据

互斥对象 mutex

mutex m;//控制共享数据访问的mutex

int sh;//共享数据

void f()

{

unique\_lock<mutex> lck{m};//获取mutex

sh += 7;//处理共享数据

}//隐形释放mutex

1. 时间功能（p102）

C++的时间功能位于<chrono>头文件中

//时间流逝

using namespace std::chrono;

auto t0 = high\_resolution\_clock::now();

this\_thread::sleep\_for(milliseconds{20});//this\_thread默认指向唯一的线程

auto t1 = high\_resolution\_clock::new();

cout<<duration\_cast<nanoseconds>(t1-t0).count()<<”nanoseconds passed\n”;

条件变量：通过外部事件实现线程间通信的基本防范是条件变量condition\_variable，它定义在<condition\_variable>中。condition\_variable提供了一种机制，允许一个thread等待另一个thread。特别是，它允许一个thread等待某个条件（condition，通常称为一个事件，event）发生，这种条件通常是其他thread完成工作产生的结果。

class Message//通信的对象

{

};

queue<Message> mqueue;//消息队列

condition\_variable mcond;//通信用的条件变量

mutex mmutex;//锁机制

consumer()读取并处理Message：

void consumer()

{

while(true)

{

unique\_lock<mutex> lck{mmutex};//获取mmutex

while(mcond.wait(lck)) /\*do nothing\*/;//释放lck并等待

//被唤醒后重新获取lck

auto m = mqueue.front();//获取消息

mqueue.pop();

lck.unlock();//释放lock

//…处理m…

}

}

对应的producer()这样编写：

void producer()

{

while(true)

{

Message m;

//…填入消息…

unique\_lock<mutex> lck{mmutex};//保护队列上的操作

mqueue.push(m);

mcond.notify\_noe();//通知

}//释放锁（在作用域结束）

}

1. future和promise（p104）

future和promise的关键点是他们允许在两个任务间传输值，而无需显示使用锁。基本思路很简单：当一个任务需要向另一个任务传输某个值时，他把值放入promise中。

如果我们有一个名为fx的future<X>,则可以使用get()得到一个类型为X的值：

packaged\_task

async()

# 2017/10/16 son

1. 时间（p106）

using namespace std::chrono;//见35.2节

auto t0 = high\_resolution\_clock::now();

do\_work()

auto t1 = high\_resolution\_clock::new();

cout<<duration\_cast<milliseconds>(t1-t0).count()<<” msec \n”;

系统时钟返回一个time\_point类型的值（时间点），两个time\_point相减的结果是duration(时间段)

1. 类型函数（p107）

编译时求值的函数，他接受一个类型作为实参或者返回一个类型作为结果。

constexpr float min = numeric\_limits<float>::min();//最小的正浮点数

constexpr int szi = sizeof(int);//int所占的字节数量

类型函数是C++的编译时计算机制的一部分，它允许程序进行更严格的类型检查以及获取更优的性能。我们通常把这种用法称为元编程或者模板元编程。

1. 类型谓词

<type\_traits>中定义了类型谓词

is\_arithmetic、is\_class、is\_pod、isliteral\_type、has\_virtual\_destructor和is\_base\_of等。

bool b1 = is\_arithmetic<int>();//YES，int是一种算数类型

bool b2 = is\_arithmetic<string>();//NO，std::string不是一种算数类型

编写模板时经常用到这些谓词，例如：

template<typename Scalar>

class complex

{

Scalar re, im;

public:

static\_assert(is\_arithmetic<Scalar>(), “sorry, I only support complex of arithmetic types”);

}

1. pair和tuple(p109)

equal\_range返回迭代器的一个pair，表示一个满足给定谓词的子序列

pair 元素个数两个

make\_pair()函数快捷地创建一个pair,而无需显示指定他的类型，如：

void f(vector<string>& v)

{

auto pp = make\_pair(v.begin(), 2);//pp的类型是pair<vector<string>::iterator, int>

}

如果元素个数不止两个（或者不足两个），则应该用tuple（定义在<utility>中，见34.2.4.2）。tuple可以表示任意形式的元素序列，如：

tuple<string, int, double> t2(“sild”, 123, 3.14);

auto t = make\_tuple(string(“Herring”), 10, 1.23);//隐式地推断出类型是

//tuple<string, int, double>

string s = get<0>(t);//获取tuple的第一个元素：“Herring”

int x = get<1>(t);

double d = get<3>(t);

要想在编译时从tuple当中选取元素，只能使用get<1>(t)的方式（尽管看起来不简洁），而不能写成get(t, 1)或者t[1]（见28.5.2）

1. 随机数（p112）

标准库在<random>中提供了很多种不同的随机数生成器。随机数生成器包括两部分：

[1]一个引擎（engine），负责生成一组随机值或者伪随机值

[2]一种分布（distribution），负责把引擎产生的值映射到某个数学分布上

常用的分布包括uniform\_int\_distribution（生成的所有证书概率相等）、normal\_distribution（正态分布，又名“铃铛曲线”）和exponential\_distribution（指数增长）

using my\_engine = default\_random\_engine;//引擎类型

using my\_distribution = uniform\_int\_distribution<>;//分布类型

my\_engine re{};//默认引擎

my\_distribution one\_to\_six{1,6};//该分布吧随机数映射到1~6的范围

auto die = bind(one\_to\_six, re);//得到一个随机数生成器

int x = die();//掷骰子：x得到的值位于1~6之间

标准库的bind生成一个函数对象，它会把第二个参数（re）作为实参绑定到第一个参数（one\_to\_six函数对象）的调用中（见33.5.1）。因此die()等价于调用one\_to\_six(re)。

简易版的随机数生成：

class Rand\_int

{

public:

Rand\_int(int low, int high):dist{low, high}

int operator()(){ return dist(re); }

private:

default\_random\_engine re;

uniform\_int\_distribution<> dist;

}

Rand\_int rnd{1, 10};//构建一个随机数生成器，生成1~10之间的随机数

int x = rnd();//x是1~10之间的随机数

1. 数值限制

static\_assert(numeric\_limits<char>::is\_signed, “unsigned characters!”);

static\_assert(100000 < numeric\_limits<int>::max(), “small ints!”);

注意，因为numeric\_limits<int>::max()是一个constexpr函数，所以第二个断言是有效的

## 第六章 类型与声明

1. 布尔值

bool b2{7};//错误：发生了窄化转换

bool b3{7!=0};

bool a = true;

bool b = true;

bool x = a+b;// a+b结果是2，因此想最终取true

bool y = a||b;//a||b结果是1，因此最终取true

bool z = a-b;//a-b结果是0，因此z的最终取值是false

如有必要，指针叶梦被隐式地转换成bool（见10.5.2.5）。其中非空指针对应true，值为mullptr的指针对应false。

1. 带符号字符和无符号字符（p122）

char类型到底带不带符号是依赖于实现的，这可能带来一些意料之外的糟糕的结果，如：

char c = 255; //255的二进制表示是全“1”形式，对应的16进制是0xFF

int i = c;

i的值是几不确定，如果运行环境中一个字节站8位，则答案依赖于char的全”1”形式在转换为int时是何含义。若计算机的char是无符号的，则答案是255；反之，若计算机的char是带符号的，则答案是-1。

虽然从本质上来说，char的行为无非与signed char一致或者与unsigned char一致，但这3个名字代表的类型的确各不相同。我们不能混用指向这3种字符类型的指针，如：

void f(char c, signed char sc, unsigned char uc)

{

char\* pc = &uc;//错误：不错在对应的指针转换规则

signed char\* psc = pc;//错误：不错在对应的指针转换规则

unsigned char\* puc = pc;//错误：不错在对应的指针转换规则

psc = puc;//错误：不错在对应的指针转换规则

}

3种char类型的变量可以相互赋值，但是把一个特别大的值赋给带符号的char是未定义的行为。例如：

void g(char c, signed char sc, unsigned char uc)

{

c = 255;//如果普通的char是带符号的且占8位，则该语句的行为依赖于具

//体实现

c = sc; //OK

c = uc;//如果普通的char是带符号的且uc的值特别大，则该语句的行为依//赖于具体实现

sc = uc;//如果uc的值特别大，则该语句的行为依赖于具体实现

uc = sc; //OK,转换成无符号类型

sc = c;//如果普通的char是带符号的且uc的值特别大，则该语句的行为依

//赖于具体实现

uc = c; //OK,转换成无符号类型

}

1. 字符字面值常量（character literal）是指单引号内的一个字符，如‘a’和‘0’等（p123）

|  |  |  |
| --- | --- | --- |
| 字符名 | ASCII | C++名字 |
| 八进制数 | ooo | \ooo |
| 十六进制数 | Hhh | \xhhh… |

我们可以把字符集中的字符表示成一个1~3位的八进制数（\后紧跟八进制数）或者表示成十六进制数（\x够紧跟十六进制数字）。其中，序列里十六进制数字的数量是没有限制。如果遇到了第一个不是八进制数字或者十六进制数字的字符，则表明当前的八进制序列或者十六进制序列已经结束。

char v1[] = “a\xah\129”;//6个字符：‘a’ ‘\xa’ ‘h’ ‘\12’ ‘9’ ‘\0’

char v4[] = “a\xad\0127”;//5个字符：’a’ ‘\ad’ ‘\012’ ‘7’ ‘\0’

C++程序可以操作Unicode等其他字符集。大字符集中的字面值常量通常表示成为4个或者8个十六进制数字，其前缀是u或者U。例如：

U‘\UFADEBEEF’

u’\uDEAD’

u’\xDEAD’

1. 整形类型

如果需要更精细地控制整数的尺寸，可以使用<cstdint>中定义的别名。

3是一个int,3U类型是unsigned int，而3L是一个long int

# 2017/10/16 son

1. 对其（p130）

alignof()运算符返回实参表达式的对其情况，例如：

auto ac = alignof(‘c’); //char的对其情况

auto ai = alignof(i);//int的对其情况

auto ad = alignof(2.0); //double的对其情况

int a[20];

auto aa = alignof(a);//int的对其情况

有时我们需要在声明语句中使用对其，但是不允许形如alignof(x+y)的表达式；此时，我们可以使用类型说明符alignas:alignas(T)，它的含义是“像T那样对齐”。例如，我们用下面的语句为一些类型X的变量留出未初始化的存储空间：

void user(const vector<X>& vx)

{

constexpr int bufmax = 1024;

alignas(X) buffer[bufmax];//未初始化的

const int max = min(vx.size(), bufmax/sizeof(X));

uninitialized\_copy(vex.begin(), vx,begin()+max, buffer);

}

# 2017/10/16 son

1. 初始化

当我们用auto关键字从初始化器推断变量时，没必要采用列表初始化的方式而且如果我们初始化器是{}列表，则推断得到的数据类型肯定不是我们想要的结果。例如：

auto z1{99};//z1的类型是initializer\_list<int>

auto z2 = 99;//z2的类型是int

vector<int> v1{99};//v1包含一个元素，该元素的值是99

vector<int> v2(99);//v2包含99个元素，每个元素都取默认值0

1. 初始化列表（p141）

在声明语句中，一对()通常表示“函数”，因此，如果想显示地表达“执行默认初始化”的意愿，你需要使用{}。如：

complex<double> z1(1,2);//函数风格的初始化器（用于构造函数执行初始化）

complex<double> f1();//函数声明

complex<double> z2;//使用构造函数初始化成{1,2}

complex<double> f2{};//用构造函数初始化成默认值{0,0}

1. decltype()修饰符

当有一个合适的初始化器的时候，可以使用auto。但是很多时候我们既想推断得到类型，又不想在此过程中定义一个初始化变量，因此，我们应该生命类型修饰符decltype(expr)。如果我么想编写一个函数灵气执行两个矩阵的加法运算，但是这两个结果矩阵的元素是什么类型？最自然的回答是：结果矩阵的元素应该是对应元素求和后的类型。因此，我们声明如下：

template<class T, class U>

auto operator+(const Matrix<T>& a, const Matrix<U>& b) ->Matrix<decltype>(T{}+U{})>

{

Matrix<decltype(T{}+U{})> res;

for(int i=0; i!=a.rows(); ++i)

for(int j=0; j!=a.cols(); ++j)

res(i, j) = a(I, j) + b(I, j);

return res;

}

# 2017/10/16 son

1. 不允许在类型别名前加修饰符(如，unsigned)，例如：

using Char = char;

using Uchar = unsigned Char;//错误

using Uchar = unsigned char;//OK

## 第七章 数组和引用

1. void\*（p149）

void f(int\* pi)

{

void\* pv = pi; //OK，发生了从int\*到void\*的隐式类型转换

\*pv;//错误，不允许解引用void\*

++pv;//错误，不允许对void\*执行递增操作（所指向的对象尺寸不定）

}

函数指针和指向类成员的指针不能被赋给void\*;

1. nullptr（p150）

int i = nullptr;//错误，i不是指针

在原来的代码中，很多人习惯定义一个宏NULL来表示空指针。例如：

int\* p = NULL;//使用宏NULL

然而，在不同的具体实现中NULL的定义有所差异；例如，NULL可能是0，或者0L。在C语言中，NULL通常是(void\*)0，这种用法在C++中时非法的。

1. （p152）在C和C++的旧代码中，允许吧字符串字面值常量赋给一个非常量char\*：

void f()

{

char\* p = "plato";//错误：但在C++11之前的代码接受

p[4] = 'e';//错误：试图为常量赋值

}

字符串字面值常量是静态分配的，因此函数返回字符串字面值常量是很安全的行为，不会有什么问题。例如：

const char\* error\_message()

{

//…

return “range error”;

}

调用error\_message后，存放“range error”的内存区域不会消失。

两个完全相同的字符串字面值常量是在同一个数组还是在两个不同的数组中依赖于实现，例如：

const char\* p = “Heraclitus”;

const char\* q = “Heraclitus”;

void g()

{

if(p == q) cout<<”one!\n”;//结果依赖于实现

//…

}

允许在字符串中间存在空字符，但是大多数程序都会忽略空字符之后的内容。例如，标准库函数strcpy()和strlen()都把字符串“Jens\000Munk”当成”Juns”处理。

1. 原始字符串（p154）

在字符串字面值常量中表示反斜线(\)或者(“)，我们需要在这些符号的前面再加一个分斜线。但是当使用标准regex库书写正则表达式时，反斜线可以表示转义字符就可能出现默写潜在的错误。如果我们想表示两个反斜线隔开的单词，可能这么写：

string s = [\\w\\\\](file://///w////w)w;//这么多反斜线，祈祷自己千万别写错

显然这种约定俗成的正则表达式太容易出错了，为了解决这个问题，c++提供了原始字符串字面值常量（raw string literal）。在原始字符串字面值常量中，反斜线就是反斜线，双引号就是双引号，上面的例子变成了：

string s = R”(\w\\w)”;//没问题，肯定不会写错

原始字符串字面值常量使用R”(ccc)”的形式表示字符序列ccc，其开头的R用于把原始字符串字面值常量和普通的字符串字面值常量区别开来。这一对括号的作用是允许我们使用非转义的双引号。例如：

R”(“quoted string”)”;//字符串的内容是“quoted string”

进一步，如果我们想在原始字符串字面值常量中加入字符序列)”怎么办呢？这个要求并不常见，不过即使真的遇到了也有解决办法。因为”( 和 )”并不是唯一的分隔符，在”(…)”的框架中我们还可以在(之前和)之后加入其它分隔符。例如：

R”\*\*\*(“quoted string containing the usual terminator (“))”)\*\*\*”

//“quoted string containing the usual terminator (“))”

规则要求，符号)后的字符序列必须与符号(前面的序列完全一致。采用这个措施，我们能处理几乎所有复杂的字符串模式了。

与普通的字符串字面值常量不同，在原始字符串字面值常量中允许出现换行（真正的换行，而非换行符）。例如：

string counts{R”(1

22

333)”};

等价于 string x{“1\n22\n333”};

# 2017/10/23 mon

1. 大字符集（p155）

前缀是L的字符串（比如L”angst”）由宽字符组成，它的类型是const wchar\_t[]。类似地，前缀是LR的字符串（比如LR“(angst)”）也是由宽字符组成的，它的类型同样是const wchar\_t[]，它属于原始字符串字面值常量。这样的字符串以字符L’\0’结束。

UTF-8是一种可变宽的编码方式：常用字符占据一个字节，不常用的字符（根据使用情况度量）占据2个字符，特别罕见的字符占据3或者4个字节。

UTF-8字符串的结尾是’\0’，UTF-16是u’\0’，UTF-32是U‘\0’。

u8“folder\\file”//UTF-8字符串

u8R”(folder\\file)”;//UTF-8原始字符串

u“folder\\file”//UTF-16字符串

uR”(folder\\file)”;//UTF-16原始字符串

U“folder\\file”//UTF-32字符串

UR”(folder\\file)”;//UTF-32原始字符串

显然，unicode字符串的最终目的是处理unicode字符，例如：

u8”The official vowels in Danish are: a, e, i, o, u, \u00E6, \u00F8, \u00E5 and y .”

输出该字符串所得的结果是：

The official vowels in Danish are: a, e, i, o, u, æ, ø, å and y .

\u之后是十六进制数是一个Unicode编码点（iso.2.14.3）。编码点独立于编码方式，事实上，在不同的编码方式下编码点的表现形式会有所不同。例如，u’0430’（斯拉夫语的小写字母’a’）在UTF-8中时2字节的十六进制D0B0，在UTF-16中是2字节的十六进制0430，在UTF-32中是4字节的十六进制00000430。这些十六进制值成为通用字符名字(universal character name)。

前缀u和R对于顺序和大小写敏感：RU和Ur都不是合法的字符串前缀。

1. 数组中的指针（p155）

在c++语言中，指针与数组密切相关。数组名可以看成是指向数组首元素的指针。

令指针指向数组的最后一个元素的下一个位置（尾后位置）是有效的，这对于很多算法非常重要。

1. reinterpret\_cast

Allows any pointer to be converted into any other pointer type. Also allows any integral type to be converted into any pointer type and vice versa.

1. 右值引用

string var(“Cambridge”);

string f();

string&r2{f()};//左值引用，错误：f()是右值

string&r3{“Princeton”};//左值引用，错误：不允许绑定到临时变量

string&&rr1{f()};//右值引用，正确：rr1绑定到一个右值（临时变量）

string&&rr2{var};//右值引用，错误：var是左值

string&&rr3{“Oxford”};//rr3引用的是一个临时变量，他的内容是“Oxford”

const string cr1&{“Harvard”};//OK，创建一个临时变量，然后把它绑定到cr1

声明符&&表示“右值引用”，我们不是用const右值引用因为右值引用的大多数用法都是建立在能够修改所引用对象的基础之上的。const左值引用和右值引用都能绑定右值，但是他们的目标完全不同：

* 右值引用实现了一种“破坏性读取”，某些数据本来需要被拷贝，使用右值引用可以优化其性能
* const左值引用的作用是保护参数内容不被破坏

右值引用所引用对象的使用方式与左值应用所引用的对象以及普通变量没什么区别，例如：

string f(string&& s)

{

f(s.size())

s[0] = toupper(s[0]);

return s;

}

有时，程序员明确知道某一对象不再有用了，但是编译器不知道这一点。例如：

template<class T>

swap(T& a, T&b)//旧式的swap

{

T tmp{a};//此时,我们拥有了两份a

a = b;//此时,我们拥有了两份b

b = tmp;//此时,我们拥有了两份tmp（即a）

}

如果T是string和vector等拷贝操作非常昂贵的类型，则上面这个swap函数会非常昂贵。

template<class T>

swap(T& a, T&b)//几乎完美的的swap

{

T tmp{ static\_cast<T&&>(a) };//初始化的同时对a写操作

a = static\_cast<T&&>(b);//赋值的同时对b写操作

b = static\_cast<T&&>(tmp);//赋值的同时对tmp写操作

}

static\_cast<T&&>(x)的结果值是T&&类型的右值引用，引用对象是x，

在swap中使用static\_cast显然有些繁琐，程序员有时还可能拼写错误，因此标准库提供了一个名为move()的函数：move(x)等价于static<X&&>(x),其中的类型是X。通过move()，我们就能让swap的形式更清晰简洁：

template<class T>

swap(T& a, T&b)//几乎完美的的swap

{

T tmp{ move(a) };//从a移出值

a = move(b);//从b移出值

b = move(tmp);//从tmp移出值

}

1. 指针与引用

如果你想自定义（重载）一个运算符，是指用于指向对象的某物，应该使用引用。例如：

Matrix operator+(const Matrix&, const Matirx&);//OK

Matrix operator-(const Matrix\*, const Matrxi\*);//错误：不是用户自定义类型参数

Matrxi y, z;

//…

Matrix x = y+z;//OK

Matrix x2 = &y-&z;//难看且存在错误

C++不允许重新定义指针等内置类型的运算符含义

如果你想让一个集合中的元素指向对象，应使用指针：

int x, y;

string& a1[] = {x, y};//错误：引用的数组

string\* a2[] = {&x, &y};//OK

vector<string&> s1 = {x, y};//错误：引用的向量

vector<string\*> s2 = {&x, &y};//OK