## 第八章 结构、联合与枚举

1. struct的布局（p175）

在内存中为成员分配空间时，顺序与声明结构的是时候保持一致。

struct Readout

{

char hour;

int value;

char seq;

};

你可以设想Readout对象在内存中的分布情况如下所示：

hour：value：seq：

|  |  |  |
| --- | --- | --- |
|  |  |  |

然而，一个struct对象的大小不一定恰好等于它所有元素大小的累积之和。因为很多计算机要求一些特定类型的对象沿着系统结构设定的边界分配空间，以便计算机能高效地处理这些对象。在4字节int机器上，Readout的布局很可能是：

|  |  |
| --- | --- |
|  |  |
|  | |
|  |  |

hour：

value：

seq：

在预测Readout所占空间大小时，很多人简单地把每个成员的尺寸加在一起，得到结果6。其实，在此例中sizeof(Readout)真正结果是12。

你可以把成员按照各自的尺寸排序（大的在前），这样能在一定程度上减少空间浪费，例如：

struct Readout

{

int value;

char hour;

char seq;

};

此时，Readout的存储方式是：

|  |  |  |
| --- | --- | --- |
|  | | |
|  |  |  |

value：

(hour,seq):

在Readout中仍有包含一个2字节的“空洞”（未使用空间），sizeof(Readout) == 8.

在结构中如果遇到多个访问修饰符（即，public、private和prptected），有可能影响布局。

1. struct的名字（p176）

类型名字只要一出现就能马上使用了，无须等到该类型的声明全部完成。例如：

struct Link

{

Link\* preious;

Link\* successor;

};

但是，只有等到struct的声明全部完成，才能声明它的对象。

为了符合c语言早期的规定，C++允许在同一个作用域中分别声明一对同名的struct和非struct。例如：

struct stat{ /\*…\*/ };

int stat(char\* name, struct stat\* buf);

此时，普通的名字（stat）默认是非struct的名字，要想表示struct必须在stat前加上前缀struct。类似地，我们还可以让class、union和enum作为名字的前缀以避免二义性。建议应该尽量避免使用这种同名实体。

1. 结构与类（p177）

如果你只想按照默认的顺序初始化结构的成员，则结构体不需要专门定义一个构造函数，例如：

struct Point

{

int x, y;

};

Point p0;//危险行为：如果位于局部作用域，则p0未初始化

Point p1{};//以默认方式构造：{{}, {}};即 {0, 0}

Point p2{1};//以默认方式构造第二个成员：{1, {}}; 即{1, 0}

Point p3{1, 2}; //{1, 2}

2017/10/24 Tue

1. 类型等价

对于两个struct来说，即使他们的成员相同，他们仍是不同的类型。

1. 普通旧数据（p180）

有时候，我们只想把对象当成“普通旧数据”（内存中的连续字节序列）而不愿考虑哪些高级语义概念，比如运行时多态、用户自定义拷贝语义等。

POD（“普通旧数据”）是指能被“仅当做数据”处理的对象，程序员无需估计类布局的复杂性以及用户自定义的构造、拷贝和移动语义。例如：

struct S0{};//是POD

struct S1{int a;};//是POD

struct S2{int a; S2(int aa):a(aa){}};//不是POD（不是默认构造函数）

struct S3{int a; S3(int aa):a(aa){} S3(){} };//不是POD（用户自定义的默认构造函数）

struct S4{int a; S4(int aa):a(aa){} S4() = default; };//是POD

struct S5{virtual void f(); /\*\*/ };//不是POD（含一个虚函数）

struct S6:S1{};//是POD

struct S7:S0{int b;};//是POD

struct S8:S1{int b;};//不是POD（数据即属于S1又也属于S8）

struct S9:S0,S1{};//是POD

如果我们想把某个对象“仅当做数据”处理（当做POD），则要求满足下述条件？

* 不具有复杂的布局（比如含有vptr）
* 不具有非标准（用户自定义的）拷贝函数
* 含有与一个最普通的默认构造函数

当一个构造函数无需执行任何实际操作时（如果需要定义一个默认构造函数，使用=default）

一个类型具有标注布局，除非它：

* 含有一个标准布局的非static成员或基类
* 包含virtual函数
* 包含virtual基类
* 包含应用类型的成员
* 其中的非静态数据成员有多种访问修饰符
* 阻止了重要的的布局优化

si\_pod是一个标准库类型属性谓词，它定义在<type\_traits>中，我们可以通过它在代码中提问：“T是POD吗？”

1. 域（p182）

看起来用一整个字节（char或者bool）表示一个二元变量（比如on/off开关）有点浪费，但是char已经是C++中能独立分配和寻址的最小对象了。我们也可以把这些微小的变量组织在一起作为struct的域（field）。域也称为位域。我们只要指定成员所占的位数，就能把他定义成域了。C++允许未命名的域。

struct PPN

{

unsigned int PEN : 22;

int : 3;

unsigned int CCA : 3;

bool nonreachable : 1;

bool dirty : 1;

bool valid : 1;

bool global : 1;

}

域必须是整形或枚举类型。我们无法获取域的地址，除此之外，域的用法和其他变量一样。

出人意料地，用域把几个变量打包到一个字节并不一定节省空间，这种做法虽然节省了数据空间，但是负责管理和操作这些变量的代码在绝大多数机器上都会更长。经验表明，当二进制变量存储方式从域变成字节时，程序规模会显著缩小！

1. 联合（p183）

union是一种特殊的struct，它的所有成员都分配在同一个地址空间上。因此一个union实际占用的空间大小与其最大成员一样.

从技术上来说，union是一种特殊的struct，而struct是一种特殊的class。然而，很多提供给类的功能与联合无关，因此对union是加了一些限制：

* union不能含有虚函数
* union不能含有引用类型成员
* union不能含有基类
* 如果union的成员含有用户自定义的构造函数、拷贝操作、移动操作或者析构函数，则此类函数对于union来说被delete掉了。换句话说，union类型的对象不能含有这些函数
* 在union的所有成员中，最多只能有一个成员包含类内初始化器
* union不能被用作其他类的基类

C++允许为联合的最多一个成员指定类内初始化器。此时，该初始化器被用于默认初始化。例如：

union U2

{

int a;

const char\* p{“”};

};

1. 匿名union

class Entry2

{

private:

enum class Tag{ nmber, text };

Tag type;

union

{

int i;

string s; //string有默认构造函数。拷贝操作及析构函数

}

public:

Entry2& operator=(const Entry2&& e)//因为存在string变量，所以必须的

~Entry2();

//…

}

在Entry2中生命的union没有命名，它是一个匿名联合（anonymous union）。匿名联合是一个对象而非一种类型，我们无需对象名就能直接访问它的成员。

Entry2中含有一个string类成员，而在string类型中有用户自定义的赋值运算符，因此Entry2的赋值运算符被delete掉了，想为Entry2的对象赋值，就必须先定义Entry2::operator=()。赋值运算兼具读写两种操作的复杂性，但是它在逻辑上与访问函数很相似：

Entry2& Entry2::operator=(const Entry2&& e)//因为存在string变量，所以必须的

{

if(type == Tag::text && e.type == Tag::text)

{

s = e.s;//常规的string赋值

return \*this;

}

if(type == Tag::text) s.~string();//显示销毁

switch(e.type)

{

case Tag::number:

i = e.i;

break;

case Tag::text:

new (&s)(e.s); //new的作用域是显示地构造string

type = e.type;

}

return \*this;

}

Entry2::~Entry2()

{

if(type == Tag::text) s.~string();//显示销毁

}

2017/10/25 Wed

1. 枚举（p188）

枚举类型用于存放用户指定的一组整数值。枚举类型的美中取值各自对应一个名字，我们把这些值叫做枚举值（enumerator）。例如：

enum class Color{ red, green, blue };

枚举类型分两种：

* enum class，它的枚举值名字(比如red)位于enum的局部作用域内，枚举值不会隐式地转换成其他类型。
* “普通的enum”，它的枚举值名字与枚举类型本身位于同一个作用域中，枚举值隐式地转换成整数。

通常情况下，简易程序员使用enum class，它很少会产生我们意想不到的结果。

1. enum class

enum class是一种限定了作用域的前类型枚举，例如：

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

enum class Warning { green, yellow, orange, red};//火警等级

两个enum的枚举值不会冲突，他们位于给咱enum calss的作用域中。

枚举常用一些整数类型表示，每个枚举值是一个整数。我们把用于表示某个枚举的类型称为它的基础类型（underlying type）。基础类型必须是一种带符号或者无符号的整数类型，默认是int。我们可以显示地指定：

enum class Warning: int { green, yellow, orange, red };//sizeof(Warning) == sizeof(int)

若果你认为上述定义太浪费空间，可以用char代替int：

enum class Warning: char { green, yellow, orange, red };//sizeof(Warning) == 1

默认情况下，枚举值从0开始，依次递增。因此，我们可以得到：

static\_cast<int>(Warning::green) == 0

static\_cast<int>(Waring::yellow) == 1

static\_cast<int>(Waring::orange) == 2

static\_cast<int>(Waring::red) == 3

在有了Waring之后，用Waring变量代替普通的int变量是的用户和编译器都能更好地理解该变量的真正用途。例如：

void f(Waring key)

{

switch(key)

{

case Waring::green:

//…相应的操作…

case Waring::orange:

//…相应的操作…

case Waring::red:

//…相应的操作…

}

}

用户很容易发现程序缺少了对yellow的处理，编译器也能发现这一点。编译器会发出一条警告信息。

我们可以用整形常量表达式初始化枚举值，例如：

enum class Printer\_flags

{

acknowledge = 1,

paper\_empty = 2,

busy = 4,

out\_of\_black = 8,

out\_of\_color = 16,

…

};

我们特地为Printer\_flags选取了一些特殊的枚举值，以便能用位运算符把它们结合在一起。enum属于用户自定义类型，因此我们可以为它定义|和&运算符：

constexpr Printer\_flags operator|( Printer\_flags a, Printer\_flags b)

{

return static\_cast<Printer\_flags>(static\_cast<int>(a) | static\_cast<int>(b));

}

constexpr Printer\_flags operator&( Printer\_flags a, Printer\_flags b)

{

return static\_cast<Printer\_flags>(static\_cast<int>(a) & static\_cast<int>(b));

}

因为enum class不支持隐式类型转换，所以我们必须在这里使用显示的类型转换。在Printer\_flags定义了|和&之后：

void try\_to\_print(Printer\_flags x)

{

if(x&Printer\_flags::acknowledge)

{

//…

}

else if(x&Printer\_flags::busy)

{

//…

}

else if(x&(Printer\_flags::out\_of\_black | Printer\_flags::out\_of\_color))

{

//缺墨：黑白或者彩色…

}

}

我们把operator|()和operator&()定义成constexpr函数，这样就能把他们用于常量表达式了。例如：

void g(Printer\_flags x)

{

switch(x)

{

case Printer\_flags::acknowledge:

//…

break;

case Printer\_flags::busy:

//…

break;

case Printer\_flags::out\_of\_black:

//…

break;

case Printer\_flags::out\_of\_color:

//…

break;

case Printer\_flags::out\_of\_black& flags::out\_of\_color:

//缺墨，黑白或者彩色

//…

break;

}

}

C++允许先声明一个enum class，稍后再给它定义。例如：

enum class Color\_code : char;//声明

void foobar(Color\_code\* p);//使用声明

//…

enum class Color\_code : char //定义

{

red, yellow,green,blue

};

一个整数类型的值可以显示地转换成枚举类型。如果这个值属于枚举的基础类型的取值范围，则转换有效；如果超出了合理范围，则转换的结果是未定义的。例如：

enum class Flag : char { x=1, y=2, z=4, e=8 };

Flag f0{};//f0的默认值是0

Flag f1 = 5;//类型错误：5不属于Flag类型

Flag f2 = Flag{5};//错误：不允许窄化转换成enum class类型

Flag f3 = static\_cast<Flag>(5);//“不近人情”的转换

Flag f4 = static\_cast<Flag>(999);//错误：999不是一个char类型的值（也许根本捕获不到）

1. 普通的enum（p191）

“普通的enum”是指C++在提出enum class之前提供的枚举类型。普通的enum的枚举值位于enum本身所在的作用域中，他们隐式地转换成某些整数类型的值。

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

enum Warning { green, yellow, orange, red };

//错误：yellow被重复定义（取值相同），red被重复定义（取值不同）

Warning a1 = 7;//错误：不存在int向Warning的类型转换

int a2 = green;//OK：green位于其作用域中，隐式地转换成int

int a3 = Warning::green;//OK

Warning a4 = Warning::green;//OK

void f(Traffic\_light x)

{

if(x == 9) {/\*…\*/}//OK,但是Traffic\_light并不包含枚举值9

if(x == red){/\*…\*/}//错误：作用域有两个red

if(x == Warning::red){/\*…\*/} //OK

if(x == Traffic\_light::red){/\*…\*/} //OK

}

普通的enum也可以指定基础类型。

enum Color\_code : char;//声明

void foobar(Color\_code\* p);//使用声明

//…

enum Color\_code : char //定义

{

red, yellow,green,blue

};

如果没有指定枚举的基础类型，则不能把它的声明和定义分开。（在vs2017上实验可以通过编译，不知道是不是书上错了）

因为普通的enum和其基础类型之间存在隐式类型转换，所以我们不需要为他专门定义运算符|。

枚举基础类型的范围（p192）。

1. 未命名的enum

一个普通的enum可以是未命名的，例如：

enum { arrow\_up = 1, arrow\_down, arrow\_sideways };

如果我们需要的只是一组整形常量，而不是用于声明变量的类型，则可以使用未命名的enum。

2017/10/26 Thu

## 第九章 语句

1. 一个声明是一条语句，表达式的末尾加上一个分号也是一条语句。分号本身也是一条语句，即空语句。

花括号（{}）括起来的一个可能为空的语句序列块称为块（block）或者复合语句（compount statement）。

赋值和函数调用不是语句，是表达式。

1. switch语句

case标签中出现的表达式碧玺石整数或者枚举类型的常量表达式。在同一个switch语句中，一个值最多被case标签使用一次

void f(int i)

{

switch(i)

{

case 2.7://错误：在case中使用浮点数

//…

case 2：

//…

case 4-2://错误：在case中是使用了两次2

//…

}

}

有一种情况下不应该使用default：switch语句希望他的每个分支对应枚举类型中的一个枚举值。如果这样的话，最好不要使用default语句，应该让编译器负责发觉并报告case分支与枚举值未能完全匹配的问题。例如，如下几乎肯定是错的：（在vs2017上实验可以通过编译，不知道是不是书上错了）

enum class Vessal{ cup, glass, goblet, chalice };

void problematic(Vessal v)

{

switch(v)

{

case Vessal::cup :

break;

case Vessal::glass:

break;

case Vessal::goblet:

break;

}

}

1. 条件中的声明

if( double d = prim(true) )

{

left /= d;

break;

}

首先声明d并给它赋初值，然后把它初始化后的值作为条件进行检查。d的作用域从声明处开始，到条件控制语句结束为止。假设还有个else分支与上面的if分值对应，则d在两个分支中都有效。

条件中的声明语句只能声明名初始化一个变量或者const

1. 范围for语句

for(int x : v) 读作“对于范围v中的每一个元素x”，或者干脆说“对于v中的每一个x”。

冒号之后的表达式必须是一个序列（一个范围），换句话说，如果我们对它调用v.begin()和v.end()或者begin(v)和end(v)，得到的应该是一个迭代器：

* 编译器首先尝试寻找并使用成员begin和end。如果找到了begin和end，但是他们不能表示一个范围（比如，begin有可能是变量而非函数），则当前的范围for是错误的。
* 如果没有找到，则编译器继续在外层作用域寻找begin/end成员。如果找不到或者找到的不能用（比如begin不接受当前序列类型的实参），则范围for是错误的。

对于内置数组T v[n]来说，编译器使用v和v+N代替begin(v)和end(v)。<iterator>头文件为内置数组和所有标准库容器提供了begin(c)和end(c)。

如果想在范围for循环内修改元素的值，应该使用元素的引用。

void incr(vector<int>& v )

{

for(int& x : v )

{

++x;

}

}

## 第十章 表达式

1. std::[istream](http://www.cplusplus.com/reference/istream/istream/)::putback（p214）

将一个字符重新放回流中

比如，当读取数字时，如果一个字符是0~9或者‘.’，则放回流中，然后读取一个数字

istream \*p;

//…

char ch = 0;

\*ip>>ch;

switch(ch)

{

case ‘0’: case ‘1’: case ‘2’: case ‘3’: case ‘4’: case ‘5’: case ‘6’: case ‘7’: case ‘8’: case ‘9’: case ‘.’:

ip->putback(ch);//把第一个数字（或者‘.’）放回到输入流中

\*ip >> ct.number\_value;//把数字读入ct

//…

}

1. isalpha（p215）

标准库函数int isalpha ( int c )，检测c是否是一个字母

读取一个名字：

default://名字，名字=，或者错误

if(isalpha(ch))

{

ip->putback(ch);//把第一个字符放回输入流中

\*ip >> ct.string\_value;//把string读入ct

ct.kind=Kind::name;

return ct;

}

检测采用的是表格查找的方式，因此使用isspace比单独检测每个空白符要快的多。

1. isdigit() 是否是数字。 isalpha() 是否是字母。 isalnum() 是否是数字或者字母

2017/10/31 Tue

1. 类型转换（p221）

dynamic\_cast<type>(expr)运行时检查的类型转换

static\_cast<type>(expr)编译时检查的类型转换

reinterpret\_cast<type>(expr)不检查的类型转换

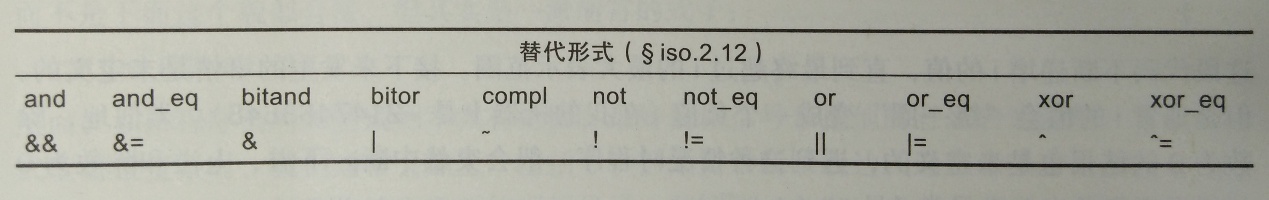
const\_cast<type>(expr)const转换

1. sizeof expr对象尺寸；sizeof(type)类型尺寸； sizeof… name 参数包尺寸；

alignof(type)类型对其；

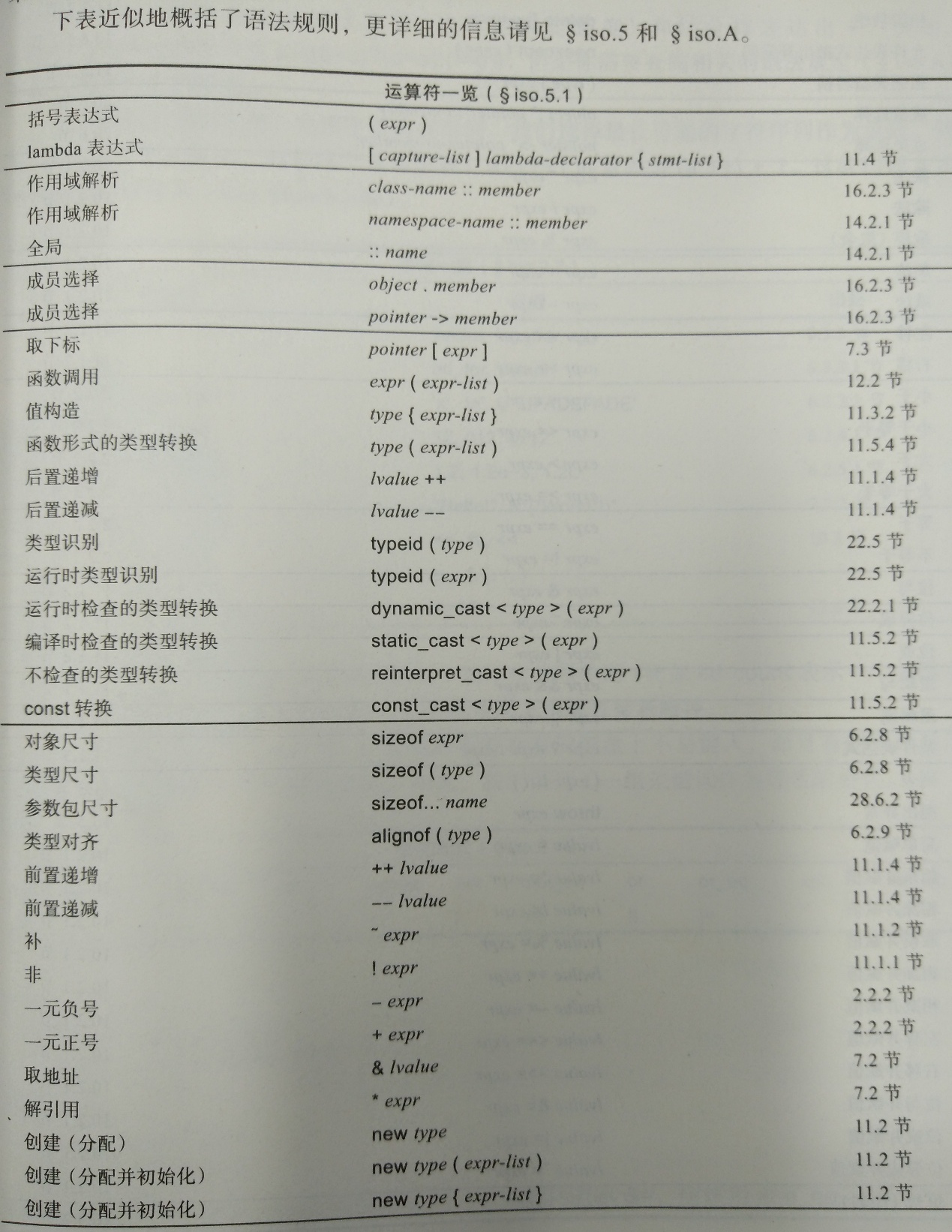
noexcep(expr)允许表达式抛出异常么；(type)expr强制类型转换；

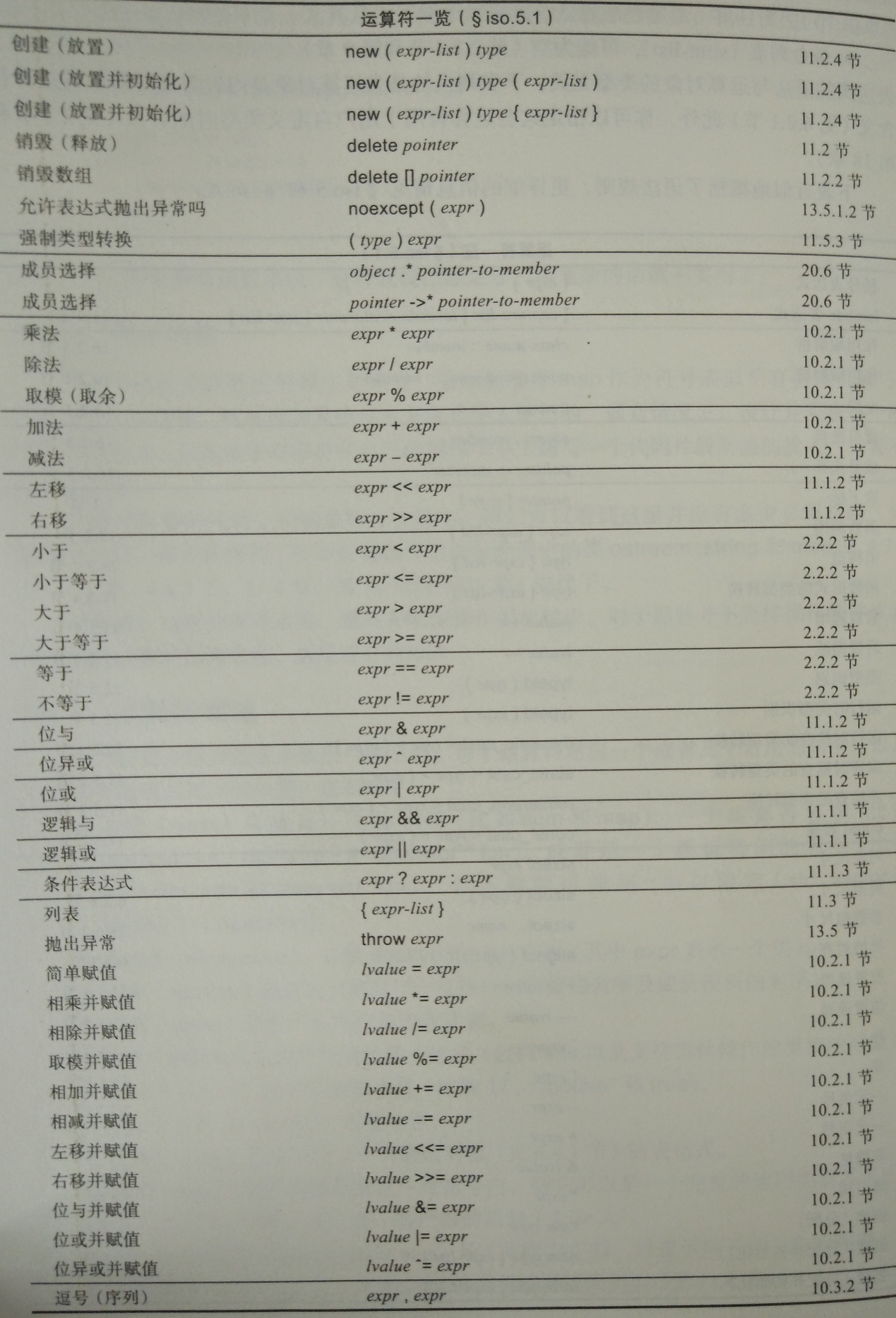
object.\*pointer-to-member成员选择；pointer->\*pointer-to-member成员选择；



1. 符号优先级

在同一个方块中的具有相同的优先级，运算符所在的方块越靠前，优先级越高





1. 求值顺序（p224）

C++没有明确规定表达式中字表达式的求值顺序，尤其请注意，你不能假定表达式是按照从左到右的顺序求值的。例如：

int x = f(2)+g(3);//到底先调用f()还是先调用g()并没有明确的规定

不限定表达式的求值顺序有助于生成更好的代码，但是有时也带来一些问题。例如：

int i = 1;

v[i] = i++;//未定义的结果

其中的赋值操作即可能执行为v[1]=1，页可能执行为v[2=1]，甚至会产生非常奇怪的运行结果。

逗号运算符（,）、逻辑与运算符（&&）和逻辑或运算符（||）规定他们的左侧运算符对象先被求值，然后是右侧运算对象。例如，b=(a=2, a+1)的意思是把3赋给b。

请注意，序列运算符（逗号）与调用函数时的分隔实参的逗号在逻辑上完全是两回事。例如：

f1( v[i], i++ );//两个实参

f2( (v[i], i++) );//一个实参

f1的调用语句包含两个实参v[i]和 i++，C++没有明确规定这两个实参表达式的求值顺序，因此这种用法尽量避免。f2的调用语句只含有一个实参，即逗号表达式，它的效果等价于i++，这种写法有一定的迷惑性，也应该尽量避免。

1. 临时对象

除非我们把临时对象绑定到引用上或者用它初始化一个命名对象，否则大多数时候在临时对象所在的完整表达式末尾，他就会被销毁了。完整表达式不是任何其他表达式的子表达式。

void f(string& s1, string& s2, string& s3)

{

const char\* cs = (s1+s2).c\_str();

cout<<cs;

if( strlen( cs = (s2+s3).c\_str() ) <8 && cs[0]==’a’ )

{

//使用cs

}

}

程序创建了一个临时的string对象保存s1+s2的结果，然后从该对象引出一个C风格字符串的指针。在表达式的末尾，该临时对象被删除。然而，c\_str()返回的C风格字符串是作为保存s1+s2结果的临时对象的一部分被分配的，我们无法确保当临时对象被销毁后，一定会退出该部分区域。因此，cs可能会指向一片被释放的存储空间，而输出操作cout<<cs是否能正常工作就完全看运气了。编译器能检测并报告很多类似地错误。

if语句的问题更微妙一些。因为保存s2+s3结果的临时对象在完整的表达式是在条件内部创建的，所以该条件会按照我们预期的方式工作。不过，在程序进入控制语句之前，这个临时变量就被销毁了，所以在受控制语句内对cs的使用无法保证有效。

请注意，就像很多其他场合一样，这个例子之所以在使用临时变量时发生问题，是因为程序以一种低层级的方式使用了高层级的数据类型。如果采用一种更清晰的程序设计风格，则不但能得到更易理解的程序片段，而且可以完全避免由临时对象带来的问题。例如：

void f(string& s1, string& s2, string& s3)

{

cout<<s1+s2;

string s = s2+s3;

if(s.length()<8 && s[0] == ‘a’)

{

//使用s

}

}

临时对象可以作为const引用或者命名对象的初始化器，例如:

void g( const string&, cosnt string& );

void h(string& s1, string& s2)

{

const string& s = s1+s2;

string ss = s1+s2;

g(s, ss);//我们可以在此使用s和ss

}

这段代码非常完美，当临时量对应的引用或者命名对象超出了作用域范围时，该临时量被销毁。切记试图返回局部变量的引用会造成程序错误，并且也不能把一个临时变量绑定到非const做引用上。

我们可以使用构造函数在表达式内部显示地创建临时对象，例如：

void f(Shape& s, int n, char ch)

{

s.move(string{n, ch});//构造一个有n个ch的拷贝的字符串，传递给

//Shape：：move()

}

1. 常量表达式（p227）

C++提供了两种与“常量”有关的概念：

* constexpr:编译时求值。
* const：在作用域内不改变其值。

常量表达式（constant expression）是指由编译器求值的表达式。他不能包含任何编译时未知值，也不能有其他的副作用。一条常量表达式由整数值、浮点数值、或者枚举值等成分构成。

C++语言要求数组的尺寸、case标签和template值实参使用常量。

int x1 = 7;

constexpr int x2=7;

constexpr int x3 = x1;//错误：初始化器不是常量表达式

constexpr int x4 = x2;//OK

1. const

以常量表达式初始化的const可以用在常量表达式中，与constexpr不同的是，const可以用非常量表达式初始化，但是此时该const将不能用作常量表达式。例如：

const int x = 7;

const string s = “asdf”;

const int y = sqrt(x);

constexpr int xx = x;//OK

constexpr string ss = s;//错误：s不是常量表达式

constexpr int yy = y;//错误：sqrt(x)不是常量表达式

发生错误的原因是string不是字面值常量类型，sqrt()不是一个constexpr函数。

2017/10/31 Tue

1. 字面值常量类型

在常量表达式中可以使用简单的用户自定义类型，例如：

struct Point

{

int x, y, z;

constexpr Point up(int d) { return {x, y, z+d}; }

constexpr Point move(int dx, int dy) { return { x+dx, y+dy, z}; }

//…

};

含有constexpr构造函数的类称为字面值常量类型（literal type）。构造函数必须足够简单才能声明成constexpr，其中“简单”的含义是它的函数体必须为空且所有成员都是用潜在的常量表达式初始化的。例如：

constexpr Point origo{0, 0};

constexpr int z = origo.x;

constexpr Point[] = { origo, Point{1,1}, Point{2,2}, orgio.move(3,3) };

constexpr int x = a[1].x;

constexpr Point xy{0, sqrt(2)};//错误：sqrt()函数不是常量表达式

自然而然地，我们可以定义constexpr函数，使其接受字面值常量类型的实参。例如：

constexpr int square(int x)

{

return x\*x;

}

constexpr int radial\_distance(Point)

{

return isqrt( square(p.x)+square(p.y)+square(p.z) );

}

constexpr Point p1{10, 20, 30};//默认构造函数是constexpr

constexpr p2{p1.up(20)};//Point::up()是constexpr的

constexpr int dist = radial\_distance(p2);

对于成员函数来说，constexpr隐含了const的意思，所以下面的写法没必要：

constexpr Point move(int dx, int dy) const { return { x+dx, y+dy, z}; }

2017/11/02 Thu

1. 当你使用constexpr时，谨记constexpr是一个关于值的概念。此时，任何对象都无法改变值或者造成其他什么影响：constexpr实际上提供了一种微型的编译时函数式程序设计语言。（p230）
2. 地址常量表达式（p231）

全局变量等静态分配的对象的地址是一个常量。然而，改地址的值是由链接器赋值的，而非编译器。因此编译器并不知道这类地址常量的值到底是什么。这就限制了指针或者引用类型的常量表达式的使用范围。例如：

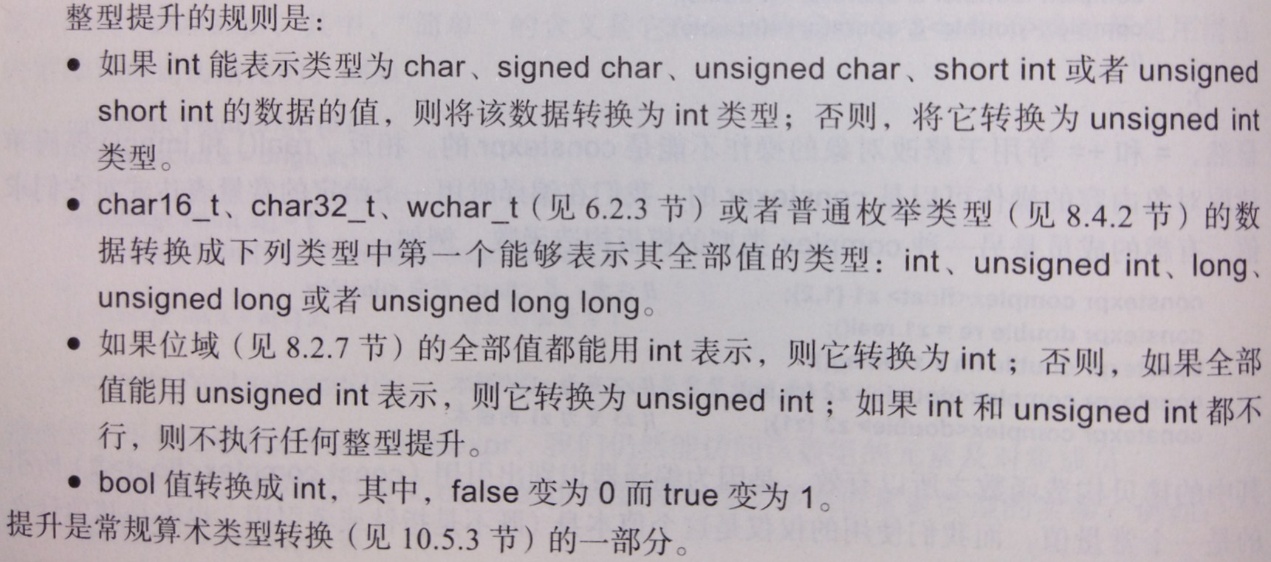
constexpr cosnt char\* p1 = “asdf”;

constexpr cosnt char\* p2 = p1;//ok

constexpr cosnt char\* p2 = p1+2;//错误，编译器不知道p1本身的值是多少

constexpr char c = p1[2];//OK， c==’d’；编译器知道p1所指的值

1. 提升（p231）



1. 类型转换（p232）

如果潜在的窄化类型转换确实无法避免，则程序员应该考虑使用一些在运行时执行检查的类型转换函数（比如narrow\_cast<>()）。

1. 浮点数类型转换（p233）

float f = FLT\_MAX;//最大的单精度浮点数

double d2 = DBL\_MAX;//最大的双精度浮点数

long double ld2 = numeric\_limits<long double>::max();

FLT\_MAX、DBL\_MAX和numeric\_limits<>定义在<limits>中。

1. 指针和引用转换（p233）

指向函数的指针和指向成员的指针不能隐式地转换成void\*。

T\*可以隐式地转换成const T\*。类似地，T&能隐式地转换成const T&。

1. 显示的检查类型转，例如：（p234）

char checked\_cast(int i)

{

char c = i;//警告：不可移除

if( c! = i ) throw std::runtime\_error{“int\_to\_char check failed”};

return c;

}

通过使用numeric\_limits，能确保截断以一种可一直的方式进行。在初始化过程中，{}初始化器形式有助于避免截断的发生。

