# 附录A

# 单定义原则

它更熟悉的称呼为ODR，单定义原则是C++编程中具有良好构建形式的基础。ODR最常见的结果在记忆与使用上足够简单：在所有文件中仅定义一次非inline函数或对象，并在每个编译单元中尽量定义一次类、内联函数和inline变量，确保同一实体的所有定义完全相同。

## A.1 编译单元

在实践中我们以写“代码”填充文件的方式编写C++程序。然而，文件设置的边界在ODR的语境中不是十分重要。相反，重要的是编译单元。本质上来说，一个编译单元是对你提供给编译器的文件使用预编译器的结果。预编译器忽略条件编译指令（#if,#ifdef,and friends）中未被选中的代码，忽略注释，插入#include中的文件（递归性地），并扩展宏。

因此，正如ODR提到的，有以下两个文件

*//====header.hpp*

*#ifdef* DO\_DEBUG

*#define* debug(x) std::cout<<x<<’n’

*#else*

*#define* debug(x)

*#endif*

void debugInit();

*//====myprog.cpp*

#include “header.cpp”

{

debugInit();

debug(main”);

}

与下面的文件等价：

*//====myprog.cpp*

void debugInit();

int main()

{

debugInit();

}

跨编译单元（边界）的连接建立在具有对应的声明和外部链接的两个翻译单元（比如，两个global函数声明debugInit()）。

注意编译单元的概念仅仅比一个“预处理文件”更抽象一点。比如，如果我们两次将同一预处理文件交给一个编译器构建一个程序，它会为该程序生成两个完全不同的翻译单元（然而，没理由这样做）。

## A.2 声明与定义

术语“定义”和“声明”在日常的“程序员对话”中常常互换地使用。然而，在ODR的上下文环境中，这些词的准确含义十分重要。

一个声明是指（通常）将一个名称导入或重导入你的程序的C++构建。一个声明也许是一个定义，取决于导入的实体与其导入的方式：

**·Namespace和namespace aliases：**namespace的声明与其aliases（别名）总是定义，尽管“定义”这个术语在此环境中不太合适，因为namespace列表的成员在之后会被“扩展”（不同于classes和enumerations）。

·**Classes、Class templates、函数、函数templates、成员函数与成员函数templates：**当且仅当声明中包含一个与其名称相关的花括号体时，声明就是定义。该规则还适用于unions、operator、成员operator、static成员函数、构造函数与析构函数、以及此类参数版本的explicit特化。

·**Enumerations：**当且仅当声明中包含用括号括起来的一列枚举。

·**Local变量与nonstatic数据成员：**这些实体常被当做定义处理，尽管这些差别无伤大雅。请注意在函数定义中声明函数的参数的行为本身就是定义行为，因为它指示了local变量，但在函数声明中的非定义的函数参数并不是整个函数的定义。

·**Global变量：**若一个声明未紧接跟随在extern关键字后或有一个初始化设定式，该Global变量的声明也是其定义。其他情况则不是定义。

·**static数据成员：**当且仅当声明出现在其所属的类、类模板的外部，或该类或类模板声明了inline或constexpr，它就是定义。

·**explicit与Local特化：**声明后跟随一个template<>或template<...>时本身就是定义，除了static数据成员的explicit特化，static数据成员模板在其含一个initializer时也是一个定义。其他的声明不是定义。那些包含类型别名（typedef 或 using），使用声明、directives、template参数声明、explicit实例化推导、static\_assets声明，等等。

## A.3 一处定义原则的细节

正如我们介绍你看附录时暗示的一样，ODR规则有很多细节。我们根据这些细节涉及的范围组织规则的约束。

### A.3.1 程序的一处定义约束

在每个程序的下列内容中只有一个定义：

·noninline函数和noninline成员函数（包括全特化的函数模板）。

·noninline变量（尤其是那些，在namespace或global范围内声明的变量，并且没有static修饰符）。

·noninline static 数据成员。

例如，以下两个翻译单元组成的的C++程序是无效的：

*//翻译单元1*

int counter;

*//翻译单元2*

int counter;*//错误：已经定义！（违反ODR）*

此规则对具有内部链接的实体并不适用（尤其是那些声明在*global*域内声明中带有*static*修饰符的或在一个namespace中的实体）因为即使这两个实体同名，它们也会被当作不同的实体处理。同样的，在未命名namespace中声明的实体也是不同的，只要它们出现在不同的翻译单元中；在C++11和之后的版本中，此类实体也默认拥有external链接。比如下两个翻译单元可以拼接进一个有效的C++程序中：

*//翻译单元1*

static int counter=2;*//与其它编译单元无关*

namespace{

    void unique()*//与其它编译单元无关*

{

    }

}

*//翻译单元2*

static int counter=0;*//与其它编译单元无关*

namespace{

    void unique()

    {

        ++counter;

    }

int main()

{

    unique();

}

此外，如果名称被上下文used，程序的前面部分一定仅有一个提到的内容（声明或定义），否则是被遗弃的constexpr if 声明的分支（仅在C++17有的特性；P263，14.6节）。

术语used在此语境中有准确意义。它暗含了程序的某处有某些种类的实体的reference引起直接的代码生成的需要。该引用可以是获取一个变量的值、调用一个函数或该实体的地址。此reference在源文件中可以是explicit，或者implicit。比如，constructor抛出异常来清除那些未被使用的内存（但已分配）时，一个new表达式会对相关的delete操作符创建一个implicit调用以解决此类问题。另一个例子由copy

constructor组成，即使它们最终optimized掉它们也要被定义。（除非语言需要他们被optimized掉，在C++17中是很常见）。Virtual函数也会隐式 used（通过使能virtual函数调用的内部结构），除非他们是纯虚函数。其它几种隐式 use也存在，但为简明起见我们省略。

有些references并未在之前的场景中构成一个use：那些出现在unevaluated的操作符中的（例如，操作符*sizeof*或*decltype*）。操作符typeid的操作在只某些情况下是unevaluated（P138,9.1.1节）。具体来说，如果一个reference作为typeid操作符的一部分，它就与之前情况的使用方式不同，除非在typeid操作符中使用的参数指明了一个polymorphic（多态）对象结尾（此对象有可能继承了virtual函数）。例如，思考下面的单一文件的程序：

*#include*<typeinfo>

class Decider{

*#if* *defined*(DYMANIC)

virtual ~Decider(){

    }

*#endif*

}

extern Decider d;

int main()

{

    char const\* name = typeid(d).name();

*return* (int)sizeof(d);

}

只有预处理器符号DYMAIC未被定义，这才是一个有效的程序。实际上，变量d并未被定义，但在sizeof(d)中对d的引用并未构成use,并且只有d是具有plolymorphic类型的object时它才是use（因为通常来说，在运行前不一定能决定polymorphic typeid操作的结果）。

根据C++标准，本节描述的约束并不需要从C++实现中得出的diagnose。编程时，它们通常被链接器当作重命名或定义缺失报告出来。

### A.3.2 翻译单元的一处定义约束

没有实体可以在一个翻译单元中被多于一次的定义。所以下面的例子在C++中是无效的：

inline void f{}

inline void f{}*//错误！重定义*

这是一个使用保护词（gurads）包围头文件的代码的主要原因：

*//====guarddemo.hpp:*

*#ifndef* GURADDEMO\_HPP

*#define* GUARDDEMO\_HPP

...

*#endif*

这个guards确保第二时间内一个头文件被*#included*，其内容以被丢弃，因此避免了class、inline实体、template等其中包含内容的重复定义。

ODR也明确指出特定实体必须在特定环境下定义。该条件适用于class类型，inline函数与inline变量。接下来的几段文字中我们复习一下细节性的规则。

一个class类型X的定义（包括structs和unions）必须先于该翻译单元中的下列任何任意一种use：

·类型为X对象的创建（例如一个变量声明或使用*new*表达式）。此类创建可以是间接的，比如一个类本身包含X类型对象被创建时。

·类型X的数据成员声明。

·对X类型的对象使用sizeof或typeid操作符。

·Explicitly或implicitly获取类型X的成员。

·使用任何类型的转换将表达式转换为X或将X的表达式转换为其他类型。或使用 implicit转换，static\_cast，dynamic\_cast将指针或引用转换为X或将X的指针或引用转 换为其他类型（void\*除外）。

·为类型X的对象赋值

·定义或调用一个具有返回值类型为X类型的参数的函数。仅仅声明此类函数不需 要类型被定义。

类型的规则也适用于类模板生成的类型X，这表明在这些情况下对应的templates必须定义与X已经定义的情况之中。此情境中创造了create points of instantiation

或者POIs（P250，14.3.2节）

Inline函数在其使用的翻译单元中必须被定义（取决于其调用或被取址的代码位置）。然而，与class类型不同的是，它们定义跟得上使用的点：

inline int notSoFast();

int main()

{

    notSoFast();

}

inline int notSoFast()

{

}

尽管这在C++中有效，一些基于旧技术的编译器实际上不会“inline”函数体未可见的函数的调用，因此预期效果可能达不到。

正如class templates，使用参数化函数声明生成的函数（一个函数或成员函数template，或class template成员函数）会生成一个实例化点。不同于class templates，对应的定义可以在实例化点之后出现。

本节说明的ODR的方方面面通常被C++编译器轻松地识别；因此C++标准要求编译器在违反其中的某个规则时指明一些诊断。参数化函数缺乏定义是例外。此情况往往不会被诊断。

### A.3.3 跨翻译单元的等价性约束

在多于一个翻译单元中定义一个确定的实体的能力也带来了出现新种类的错误的潜在性：多个定义并不匹配。不幸的是，在一次处理一个翻译单元的传统编译技术下，此类错误很难察觉。因此，C++标准不强制执行检查或诊断到的多定义的区别（当然允许这样做）。如果此跨翻译单元的的约束被违反，不同的是，C++标准将认定它会导致未定义行为，这意味着任何合理与不合理的行为都有可能发生。典型的例子是，这种未诊断的错误会导致程序崩溃或错误结果，原则上来说它们也会导致其他更为直接的某种损害（例如文件损坏）。

跨翻译单元的约束指明，一个实体被定义在两处时，那两处必须由完全相同的符号序列组成（the keywords，operators，identifiers，等等在预处理后保留的内容）。此外，这些符号在各自的上下文中有相同的语义（比如identifiers可能提及到相同的变量）。

考虑下面的程序：

static int counter=0;

inline void increaseCounter()

{

    ++counter;

}

int main()

{

}

*//====translation unit2:*

static int counter=0;

inline void increaseCounter()

{

    ++counter;

}

此例有错。因为在两个翻译单元中，尽管inline函数increaseCounter()的符号序列相同，它们还是包含了代表两个不同实体的符号counter。实际上，因为两个名称为counter的变量有internal链接（static标识符），尽管他们名称相同，他们也是两个变量。无论inline函数是否被真正使用，这依旧是一个错误。

在定义被需要时，将在头文件中能在多个翻译单元中被定义的实体#included，可以保证该符号序列在几乎所有情况下不同。使用这种方法，相同的符号对应不同的东西的情况会很少发生，但这种情况发生时，它导致的错误十分神秘且难以追踪。

跨翻译单元的规则不仅适用于可以多处定义的实体，也适用于在生明中的默认参数。换一种说法，下面的程序存在未定义行为：

*//====translation unit1*

void unused(int = 3);

int main()

{

}

*//====translation unit2*

void unused(int 4);

在这里应该注意符号流的等价性有时会引入一些细微而implicit的影响。

下面的程序是C++标准提出的：

*//====translation unit1*

class X{

    public:

    X(int,int);

    X(int,int,int);

};

X::X(int,int = 0)

{

}

class D{

    X x = 0;

}

D d1;*//调用D（）的X(int,int)*

*//====translation unit2*

class X{

    public:

    X(int,int);

    X(int,int,int);

};

X::X(int,int = 0,int = 0)

{

}

class D : public X{

    X x = 0;

}

D d2;*//调用D（）的X(int,int,int)*

此例中，问题出现的原因是class D 隐式生成的构造器在两个翻译单元中是不同的。一个调用X的两参构造器，另一个调用三参构造器。如果还有别的内容，那就是限制程序中的某处默认参数行为的额外鼓励（如果可能的话，该位置应该在头文件的某处）。幸运的是，将默认参数放出类定义之外很少发生。

还有一处该规则例外说相同的符号一定代表相同的实体。如果相同的符号具有代表着相同值以及未被使用的最终结果的表达式的无关的约束（不仅是一个暗含着绑定一个提供该约束的变量reference），那个符号也是被考虑为等价的。

该例外允许下列结构的程序编译：

*//====header.cpp*

*#ifndef* HEADER\_HPP

*#define* HEADER\_HPP

int const length = 10;

class MiniBuffer{

    char buf[length];

    ...

};

*#endif //HEADER\_HPP*

原则上，这个头文件被两个不同的翻译单元included时，因为const在语境中暗示了static，编译器会创建两个不同的名为length 的constant变量。此类常量意味着定义编译时的constant值，不是运行时的某个特定存储位置。所以，只要不强制要求此类存储位置要存在（通过引用变量的地址），对于两个拥有相同值的行为已经足够。

最后时一点关于templates的记录。Templates的名称绑定发生于两阶段。*Nondependent names绑定与template*定义之处。等价性原则也适用于相似的nontemplates定义。关于实例化时名称的绑定，等价性原则必须在该处使用，绑定也必须是等价的。

# 附录B

# 值类别(value category)

表达式是C++语言的基石，它借以表达计算过程提供了主要方法。每个表达式都有一个类型，它描述了在计算产生的值的static类型。表达式7的类型为int，表达式5+2也是如此，还有表达式x，只要x是int类型的变量。每个表达式也有一个值类别，它描述一些值的构造方式中的细节以及这些细节如何影响表达式的行为。

## B.1传统的左值和右值

历史上只有两种值类别：左值（lvalues）和右值(rvalues)。左值是一个指向存储于内存中或（及其）寄存器内的值的表达式，例如是变量名为x的表达式x。这些表达式可以修改并允许其更新存储的值。例如，如果x是int类型的变量，下面的赋值会将x的值替换为7：

x = 7;

术语*lvalue*衍生于那些在可以赋值操作中充当各种角色的表达式：字母”l”代表”left-hand side”（左手边）因为（历史性原因，在C中）只有左值可以出现在赋值号的左边。相反的右值（“r”代表”right-hand side”）只可以出现在赋值表达式的右边。

然而，在当C在1989年被标准化时事情发生了改变：int const依然是被存储在内存中的值，它却不能出现在赋号的左侧：

int const x;*//x 是一个不可变左值*

x=7;*//错误：左侧需要一个可变值*

C++在此类操作上更改更多：Class右值可以出现在赋值号的左侧。此类赋值实际上是对class中合适的赋值操作符的函数调用，而不是‘简单的’数量类型上的赋值，所以它们适用于成员函数调用的（分离的）规则。

因为这些变化，术语lvalue现在有时代表*localizable*值。含有一个值的表达式不是仅有的一种左值表达式了。其它类型的表达式比如指针解应用操作，意味着该指针指向的一个存储在该地址的值，和指向class成员对象的表达式（p->data）都是左值。即使调用函数的返回值是“traditional”左值引用类型并以&声明，它也是左值。例如（细节位于679页，B.4节）：

std::vector<int> v;

v.front();*//因其返回类型是左值引用，产生一个左值*

可能意想不到的的是，string字面值也是（不可更改的）左值。

右值是不需要进行任何相关存储的纯数学值（例如7或字符’a’）。它们存在的目的是进行数学计算，一旦使用后就不能被引用。特别是除了string字面值（例如7，’a’，true，nullptr）之外的任何字面值都是右值，作为内置类型数学计算的结果（x+5,x就是Integer类型）和调用以值为返回结果的函数。也就是，所有的临时变量都是右值（这在已经引用他们且已命名的引用不适用。）

### B.1.1 左值和右值的转换

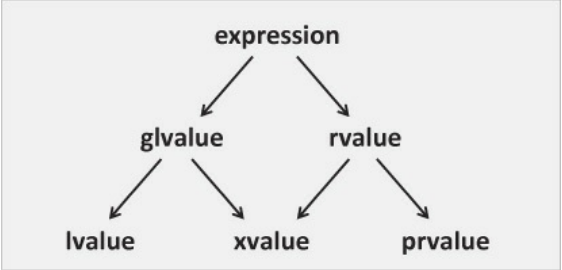
由于它们短生的本质，右值很有必要限制在一个（简单）赋值的右边：赋值7=8不合理，因为数字意义上7不允许被重定义。左值，在另一方面，并未表现出又相同的约束：当x和y是可计算类型的变量时，确实可以计算出赋值x=y，即使x和y都是左值。

赋值x=y有效，因为右边的表达式，y，经过了一个称为lvalue-to-rvalue conversion的隐式转换。正如其名，左值到右值转换以左值为参数，通过读取该左值相关的内存或寄存器生成一个相同类型的右值。因此该转换完成了两件事：首先，它确保了在预料一个右值时一个左值可以被使用（例如，作为赋值操作符的右侧或在一个数学表达式中，例如x+y）。第二，它定义了编译器可能在程序的何处发出一个“加载”指令来读取内存中的值。

## B.2 C++11值的类别

当C++11以引入右值引用以支持有移动操作的场景时，将表达式划分为左值和右值的传统方式不足以描述所有C++11的语言行为。C++标准委员会因此根据三个核心和两种复合类别重新设计了值类别系统。核心类别有：lvalue，prvalue（纯右值）和xvalue（消亡值）。复合类别有：glvalue（广义 lvalue，是lvalue和xvalue的联合）和rvalue（xvalue和prvalue的联合）。

注意所有的表达式依然是左值或右值，但rvalue类别划分的更为细致。



图片 B.1. C++11后的值类别

此C++11值类别依然有效，但在C++17中类别特性已根据以下规则重组织：

·***glvalue***（泛左值）是其赋值决定其是一个具有对象，bit-field或函数特性的表达式（例如，一个拥有存储空间的实体）。

·***prvalue***（纯右值）是其赋值初始化一个对象，bit-fied或计算一个操作符的操作数的表达式。

·***xvalue***（消亡值）是一个分配具有可被复用资源的对象或bti-feild的glvalue（通常因为名词“expire”--xvalue中的“x”来自于“eXpiring value”）.

·***lvalue***（左值）是非xvalue的glvalue。

·***rvalue***（右值）是prvalue或xvalue的表达式。

注意在C++17（以及C++11和C++14的扩展）中，讨论性上glvalue和prvalue的二分法比传统的lvalue和rvalue划分更为基础。

尽管这些描述了C++17引入的特性，那些描述依然对C++11和C++14使用（与前一个描述是等价的但更难被质疑）。

除了bit-fields，glvalues提供实体的地址。该地址可能是一个大一些的封闭对象的子对象。在基类子对象的情形中，glvalue（表达式）的类型被称为它的*static type*，最终派生对象的类型中基类是其一部分，被称为*dynamic type*的glvalue。若glvalue未提供一个基类的子对象，它的static和dynamic类型是相同的（表达式的类型）。

***lvalue***的例子有：

·指定变量或函数的表达式

·内置一元操作符\*的使用（“指针间接使用”）

·仅为string字面值的表达式

·对一个具有左值引用类型返回值类型的函数的调用。

***prvalue***的例子有：

·由字面值组成的非string字面值或用户定义字面值的表达式

·内置一元操作符&的使用（获取一个表达式的地址）

·内置算数操作符的使用

·对具有非引用类型的返回值类型的函数的调用

·Lambda表达式

***xvalues***的例子有：

·对具有返回值类型为对象类型右值引用的函数的调用（例如std::move()）

·右值引用转换为对象类型的强制转换

函数类型的右值引用提供lvalues而非xvalues。

值得强调的是glvalues，prvalues，xvalues，等等，都是表达式而非值或实体。例如，一个变量不是表达式，即使它通过表达式代表一个lvalue的变量。

int x = 3;*// 这里的x是变量而非左值。3是初始化x的纯右值。*

int y = x;*// 这里x是左值。*

*//该左值表达式的赋值并不产生值3，但指定了一个包含值3的对象。*

*//然后该左值转换为纯右值才初始化y。*

### B.2.1 临时实体化

我们之前提到过左值通常会经历一个左值到右值的转换因为纯右值是用来初始化对象的表达式（或为大多数内置操作符提供操作数）。

C++17中，此类转换有部分，被称为临时实体化（也被称为“纯右值-消亡值（xvalue）转化”）：任何时候只要一个纯右值有效的出现并碰到一个泛左值，一个临时对象会被创建并用该纯右值初始化（回忆一下纯右值是主要的“初始化值”），然后纯右值被一个指定的临时变量的消亡值替换。例如：

int f(int const&);

int r=f(3);

因为此例中的f()有一个引用参数，它需要一个泛左值参数。然而，表达式3是一个纯右值。“临时实体化规则”就此介入，表达式3被“转化”为消亡值并分配一个使用3初始化的临时对象。

更通常来说，以下的情况中，一个临时变量在实体化时会使用一个纯右值将其初始化：

·绑定于一个引用的纯右值（上例中的调用f(3)）

·获取到纯右值类型的一个类成员

·一个下标化的纯右值数组

·一个已转换为其第一个元素的指针的纯右值数组（数组衰退）

·一个出现在括起来的初始化列表中的纯右值，例如使用某些类型X，初始化类型为

std::initializer\_list<X>的对象。

·应用于纯右值的sizeof 或 typeid操作符

·一个最高级别的并声明为“expr;”形式的纯右值；或一个转换为void的表达式。

因此，C++17中纯右值初始化的对象由上下文决定，并且因此，临时变量只在被需要时创建。

早于C++17的版本中，纯右值（特别是class类型）总意味着一个临时变量。此类临时变量的复制可以有选择地在后面删除，但编译器依然必须执行复制操作的所大部分语义限制（例如一个copy构造器必须是可调用的）。接下来的例子展示了一系列C++17中该规则的修正的结果：

class N{

    public:

    N();

    N(N const&) = delete;*//此class不可复制，亦不可移动。*

    N(N&&) = delete;

};

N make\_N(){

*return* N{};*//在C++17之前总会创建一个概念性的临时变量*

}*//在C++17中，此处不会创建临时变量*

auto n = make\_N();*//C++17前为错误因为纯右值需要一个概念性副本*

*//自C++以来OK。因为n现在直接从纯右值初始化*

早于C++17时，纯右值N*{}*生成一个类型N的临时变量，但编译器可以删除副本和移动该临时变量（实际上中他总会这么做）。此情况中，这意味着调用make\_N()的临时结果可直接在n的存储中直接构造；删除和移动操作不再需要。不幸的是，C++17之前的编译器依然要检查该副本能否产生赋值或移动操作符，在这个例子中不可能，因为N的copy构造器已经删除了（并且没有move构造器生成）。因此，C++11和C++14编译器必须为此情况指明错误。

C++17中纯右值N本身不能产生一个临时变量。取而代之的是，它通过上下文初始化一个对象：在我们的例子中，是n表示的对象。没有考虑过复制或移动操作（这不是优化，而是语言保证），因此在C++17中有效。

我们可以在展示多种值类型情境的例子下做出总结：

class X{

};

X v;

X const c;

void f(X const&);*//接收一个任何值类别的表达式*

void f(X&&);*//接收纯右值和消亡值之一，取决于更匹配的值*

*//对上文中的声明*

f(v);*//给f()传递一个可更改的左值*

f(c);*//给第一个f()传递一个不可更改的左值*

f(X());*//给第二个f()传递一个纯右值*

*//（从C++17开始实例化为消亡值）*

f(std::move(v));*//给第二个f()传递一个消亡值*

## B.3使用decltype检查值类别

使用关键词decltype（引进于C++11）后，检查任何C++表达式的值类别都是可能的。对于任何表达式x，decltype((x))（注意双括号）生成：

·*type*如果x是纯右值

*·type&*如果x是左值

·*type&&*如果x是消亡值

decltype((x))的双括号用来避免在表达式x不是一个实体的名称时产生一个有名称实体的已声明类型（其他情况下此括号无效）。例如，例如，表达式x命名了变量v，没有外括号的构造变成decltype(v)，它产生变量v的类型而非反映表达式x相关变量的值类别。

因此，为任何表达式e使用类型特征，我们用如下方法检查它的值类别：

*if* constexpr(std::is\_lvalue\_reference<decltype((e))>::value){

    std::cout << "expression is lvalue\n";

}

*else* *if* constexpr(std::is\_rvalue\_reference<decltype((e))>::value){

    std::cout << "expression is xvalue\n";

}

*else*{

    std::cout << "expression is prvalue\n";

}

相关细节位于15.10.2节。

## B.4引用类别

C++中的引用类型——例如int&——以两种重要方式与值类别交互。第一种是引用可能限制表达式可绑定的值类别。例如，一个类型为int&的non-const左值引用只能被类型为int的左值的表达式初始化。相似的，一个类型为int&&的non-const右值引用只能被类型为int的右值的表达式初始化。

值类别和引用交互的第二种方式是在函数的返回类型中，使用引用类型作为返回值的类型影响调用该函数时的值类别。特别的：

·调用一个返回值为左值引用的函数产生一个左值。

·调用一个返回值为对象类型的右值引用的函数产生一个消亡值（函数类型的右值引用总会导致左值）。

·调用一个返回值为非引用类型的函数产生一个纯右值

我们用以下例子来解释引用类型与值类别的互动，给出：

int& lvalue();

int&& xvalue();

int prvalue();

值类别和给定表达式的类型可以通过decltype决定。正如15.10.2节描述的，当表达式是左值或消亡值时它用引用类型描述：

std::is\_same\_v<decltype(lvalue()),int&>;*//生成true因为结果是左值*

std::is\_same\_v<decltype(xvalue()),int&&>;*//生成true因为结果是消亡值*

std::is\_same\_v<decltype(prvalue()),int>;*//生成true因为结果是纯右值*

因此，下列调用是可能的：

int& lref1 = lvalue();*//OK:左值引用可以绑定一个左值*

int& lref3 = prvalue();*//ERROR:左值引用无法绑定一个纯右值*

int& lref2 = xvalue();*//ERROR:左值引用无法绑定一个消亡值*

int&& rref1 = lvalue();*//ERROR:右值引用无法绑定一个左值*

int&& rref2 = prvalue();*//OK:右值引用可以绑定一个右值*

int&& rref3 = xvalue();*//OK：右值引用可以绑定一个消亡值*

# 附录C

# 重载解析

重载解析是为给定调用表达式进行选择要调用函数的过程。考虑下面的简单例子：

void display\_num(int);*//#1*

void display\_num(double);*//#2*

int main()

{

    display\_num(399);*//#1比#2更匹配*

    display\_num(3.99);*//#2比#1更匹配*

}

在此例中，名为display\_num的函数成为被重载的。当该名称在一个调用中被使用时，C++编译器必须因此使用额外信息分辨多个候选者；绝大多数情况下，这些信息是调用参数的类型。在我们的例子中，它让我们直观的感觉到，当我们使用整型参数调用函数的会调用int版本并且当一个浮点数参数被提供时调用double版本。尝试将这种直觉性的选择塑模的形式化过程九四重载解析的过程。

引导重载解析并隐藏于该规则之下的常规思想十分简单，但在C++标准化过程中它的细节变得十分复杂。复杂性大都由想要支持多种现实世界中直觉性的（对人来说）看起来是一个“明显的最佳匹配”的欲望驱动，但尝试形式化这个直觉时，冒出了各种各样的子主题。

在此附录中，我们为重载解析提供合理地、细节性的描述。然而，本过程如此，如此我们也不会声明去涉及该主题的每个部分。

## C.1 何时应用重载解析？

重载解析只是一个完整的函数调用的一部分。实际上，它并不是每个函数调用的一部分。第一，通过函数指针和成员函数指针的调用就不是重载解析的主题，因为调用的函数完全取决于指针（在运行时）。第二，函数式宏无法被重载，因此它也不是重载解析的主题。

在很高层面上，对有名称函数的调用可以用以下方式处理：

·名称被检查并用于构建一个重载集

·如果有必要，该集以多种方式调整（例如，模板参数推导和替代的出现，即那些可以造成某候选模板被丢弃的行为）。

·任何完全不匹配该调用的候选（模板）从重载集中消除（即使考虑了隐式转换和默认参数之后）。该行为导致一系列的可行函数候选者。

·重载解析被执行用于寻找最佳候选者。若有，则他被选中，否则，结果是不明确的。

·被选中的候选者被检查。例如，如果其是一个已被删除的函数（例如被定义为 = delete）或无法获取的私有成员函数，会指出一个诊断。

上面的每一步都有其副标题，但重载解析可以认为是最复杂的。幸运的是，少量简洁的原则

阐明了大多数情况。我们接下来检验这些原则。

## C.2 简化的重载解析

重载解析通过比较可行候选函数的每个参数与函数调用的每个参数的匹配程度将其分级。对于一个被认定优于其它的候选者，更优候选者的每个参数在对应参数上不能比其它候选者更不匹配。下例解释以上内容：

void combine(int, double);

void combine(long, int);

int main()

{

    combine(1,2);*//有歧义！*

}

此例中，对combine()的调用有歧义，因为第一个候选者最匹配第一个参数（int类型字面值1），同时第二个候选者最匹配第二个参数。我们可以辩解int在某些感觉上相比double而言更接近long（支持选择第二个候选者）。但C++并未在参与多调用参数情况下尝试定义的接近性衡量。

给出第一个原则，我们只剩下在可行候选者中确认其对应参数与给定参数的匹配程度。作为第一次细化，我们可以用以下规则分级可能的匹配程度（最佳到最差）：

1.完美匹配。参数有表达式的类型，或者它有表达式类型的引用的类型（可能加上了const和/或 violate修饰符）。

2.小的更改后匹配。这包括，例如，数组变量腐化为其指向第一个元素的指针或const的加法去匹配一个类型为Int\*\*的参数和一个类型为int const\* const\*的参数。

3.提升后匹配。提升是一种隐式转换，包括将小整型类型（bool,char,short 有时是enumerations）转换为int,insigned int,long，或unsigned long，以及float到double的转换。

4.只在转换后匹配。包括任何类型的标准转换（Int to float），或将衍生类转换为它的public，无歧义基类，此情况不包括转换操作符和转换构造器的隐式调用。

5.与用户定义的转换匹配。该情况下允许任何隐式转换。

6.与省略符匹配（...）。一个省略符几乎可以匹配任何类型。然而有一个例外：具有非常规copy构造器的函数类型或可能无效的了悉尼港（实现可以允许也可以不允许）。

下面人为的例子解释了那些匹配的情况：

int f1(int);*// #1*

int f1(double);*// #2*

f1(4);*// 调用 #1 : 完美匹配 (#2 需要标准转换)*

int f2(int);*// #3*

int f2(char);*// #4*

f2(true);*// calls #3 : 提升后匹配*

*// (#4 需要更强的标准转换)*

class X {

public:

X(int);

};

int f3(X);*// #5*

int f3(…);*// #6*

f3(7);*// 调用 #5 : 与用户定义转换匹配*

*// (#6 需要一个与省略号的匹配)*

注意重载解析出现在模板参数推导之后，该推导并未考虑所有种类的转换。例如：

template<typename T>

class MyString {

public:

MyString(T const\*);*// 转换构造器*

};

template<typename T>

MyString<T> truncate(MyString<T> const&, int);

int main()

{

MyString<char> str1, str2;

str1 = truncate<char>("Hello World", 5);*//OK*

str2 = truncate("Hello World", 5);*//ERROR*

}

在转换构造器中提供的隐式转换在模板参数推导期间并未被考虑。对strr2的赋值并未发现可用的函数truncate()；因此重载解析总的来说并未执行。

在模板参数推导的环境中，也回想到，如果对应参数是一个左值或是以后个右值参数的右值引用类型，模板的右值引用可推导出一个左值引用类型（在引用折叠后）。例如：

template<typename T> void strange(T&&, T&&);

template<typename T> void bizarre(T&&, double&&);

int main()

{

strange(1.2, 3.4);*// OK: T被推导为double*

double val = 1.2;

strange(val, val);*// OK: T被推导为double&*

strange(val, 3.4);*// ERROR: 解析冲突*

bizarre(val, val);*// ERROR: 左值变量不匹配double&&*

}

之前的规则只是第一次细化，但它们涵盖了很多情况。虽然还有很少一些情况未被此规则充分说明。我们以对最这些规则最重要的精进的简短讨论继续。

### C.2.1 成员函数的隐式参数

非静态成员函数的调用在该成员函数的定义中，正如\*this，有一个可获取的隐藏参数。对于类MyClass的一个成员函数来说，该隐藏参数的类型为MyClass&（对于non-const成员函数）或MyClass const&（对于const成员函数）。这或许是令人惊讶的结论，this有一个指针类型。然而，在引用类型是C++语言的一部分之前，this是早期C++的一部分，当引用类型被添加时，大多数代码早已依赖于this是个指针了。

隐藏的\*this参数以与显式参数相似的方式参与重载解析。大多数时候这种方式很自然，但偶尔会变得不可预测。下例展示了一个String-like类并未如期运行（我们曾在现实中见到过此类代码）：

*#include* <cstddef>

class BadString {

public:

BadString(char const\*);

*// 通过下标获取字符:*

char& operator[] (std::size\_t);

*#1*

char const& operator[] (std::size\_t) const;

*//隐式转换为零结尾字符串（C字符串）:*

operator char\* ();

*#2*

operator char const\* ();

};

int main()

{

BadString str("correkt");

str[5] = 'c';*//可能有重载解析歧义*

}

开始，表达式str[5]看起来并无歧义。#1处的下标符看似完美匹配。然而，它不是十分完美，因为参数的5类型为int，操作符期望获得一个无符号整数类型（size\_t和std::size\_t通常类型为unsigned int 或 unsigned long,但绝不是int）。同样，一个简单的标准整数转换很容易让#1可行。然而，还有另一个可行候选者：内置下标操作符。实际上，如果我们将隐式转换操作符用于str（也会是隐式成员函数参数），我们会获得一个指针类型，现在内置下标操作符应用了。内置操作符获取一个类型为ptrdiff\_t类型的参数，在许多平台上与int等价因此完美匹配参数5。故即使内置下标操作符对于隐式参数来说是一个较差的匹配（通过用户定义转换），但它依然比#1处为实际下标定义的操作符更匹配。因此是隐式的歧义。为方便解决此类问题，你可以声明带有ptrdiff\_t参数的操作符[]，或者使用显示转换替换char\*的隐式类型转换（无论如何通常都很推荐）。对于一个可行候选者集来说，同时含有静态和非静态成员是可能的。比较静态成员和非静态成员时，隐式参数的匹配程度质量会被忽略（只有非静态成员具有隐式\*this参数）。

默认上，非静态成员具有的隐式\*this参数是左值引用类型，但C++11引入的分析变为右值引用。例如：

struct S {

void f1();*// 隐式 \*this 参数是一个 左值引用(看下面)*

void f2() &&;*// 隐式 参数是一个 右值引用*

void f3() &;*// 隐式 \*this 参数是一个左值引用*

};

正如在此例中分辨的，不仅可以将隐式参数变为右值引用（加上&&后缀）也可以断言左值引用的情况（加上&后缀）。有趣的是，明确&后缀并不等价于完全改变其原类型：一个老的特例允许一个右值绑定在一个non-const类型的左值引用上如果该引用时传统的隐式\*this参数，当左值引用的处理被明确要求时，该特例（有些危险）不再适用。所以，根据之前S的定义：

struct S {

void f1();*// 隐式 \*this 参数是一个 左值引用(看下面)*

void f2() &&;*// 隐式 参数是一个 右值引用*

void f3() &;*// 隐式 \*this 参数是一个左值引用*

};

int main()

{

S().f1();*// OK: 旧规则允许S()隐式匹配*

*// 左值引用类型的S&的 \*this*

S().f2();*// OK: 右值 S() 匹配右值引用类型的 \*this*

S().f3();*// ERROR: 右值 S() 无法匹配显式左值*

*// 引用类型的 \*this*

}

### C.2.2 优化最佳匹配

对于类型为X的参数，有四种常见参数类型构成完美匹配：X，X &，X cost&和X&&（X const&&也是一个准确匹配，但很少使用）。然而更常见的是在两种引用上重载函数。早于C++11时，这意味着以下情况：

void report(int&);*// #1*

void report(int const&);*// #2*

int main()

{

*for* (int k = 0; k<10; ++k) {

report(k);*// 调用 #1*

}

report(42);*// 调用 #2*

}

这里，没有额外const的版本适用于左值，反之只有带有const版本的可以匹配右值。

因为C++11中引进了新的右值引用，另一个常见的需要分辨的两个最佳匹配的情况在下例中得到解释：

struct Value {

…

};

void pass(Value const&);*// #1*

void pass(Value&&);*// #2*

void g(X&& x)

{

pass(x);*// 调用 #1 , 因为 X是左值*

pass(X());*// 调用 #2 , 因为 X()是右值*

(in fact, prvalue)

pass(std::move(x));*// 调用 #2 , 因为 std::move(x)是右值 (实际上是启用值)*

}

这次，具有右值引用的版本被认为右值的较优匹配，但它不能匹配左值。

注意这也对成员函数调用的隐式参数有效：

class Wonder{

  public:

    void tick();*//#1*

    void tick() const;*//#2*

    void tack() const;*//#3*

};

void run(Wonder& device)

{

        device.tick();*//调用#1*

        device.tack();*//调用#3，因为无 non-const版本*

*//的 Wonder.tack()*

}

最后，接下来对我们早期版本的修改说明了如果你在重载时使用和未使用引用，两个最佳匹配也可以造成歧义：

void report(int);*//#1*

void report(int&);*//#2*

void report(int const&);*//#3*

int main(){

*for*(int k = 0;k < 10;++k){

        report(k);*//歧义：#1 和 #2 匹配程度相同*

    }

    report(42);*//歧义：#1 和 #3 匹配程度相同*

}

## C.3 重载的细节

前面几节覆盖了日常C++编程中碰到的重载的情况。然而，不幸的是，更多的关于这些规则之外的规则和例外--出现在一本不是关于C++函数重载的书中不仅是有必要的。尽管如此，此次我们只在某种程度上讨论其中的一部分，因为他们的使用比其他的规则有些更为频繁并且有意的提供一种这些细节有多深入的感觉。

### C.3.1 优先非模板或更加专用的模板

当重载解析的其它所有方面是等价时，一个nontemplates函数比一个模板的实例更易采用（无论该实例生成于原始模板定义或是否提供了一个explicit特化）。例如：

template<typename T> int f(T);*//#1*

void f(int);*//#2*

int main(){

*return* f(7);*//错误：选择 #2，该函数无返回值*

}

该例也清晰地说明了重载解析通常不考虑所选函数的返回类型。

然而，当其他重载解析层面稍微不同时（比如有const上的不同和引用修饰符），首先常规的重载解析被执行。该行为通常会意外的引发令人吃惊的行为，当成员函数的被定义为接受相同参数作为copy或move constructor。详见333页16.2.4节。

如果在两个模板之间做出选择，那么最特化的模板更优先（假设一个模板实际上比另一个模板更特化）。该概念的详细解释位于330页的16.2.2节。该区别的一个特例是当两个模板只在尾随参数包上有区别：如果与调用匹配，没有该包的模板被认为更特化，因此更优先。57页的4.1.2节讨论了一个该情况的例子。

### C.3.2 转换序列

一个隐式转换在通常上来说可以是一系列的基本转换。考虑下面的代码例子

class Base {

public:

operator short() const;

};

class Derived : public Base {

};

void count(int);

void process(Derived const& object)

{

count(object);*// 与用户定义版本匹配*

}

调用count(object)有效因为object可以隐式转换为int。然而，该转换需要几步：

1.一个从Derived const的object转换为Base const（这是一个常左值转换，它保留了该object的独立性）。

2.导致Base const object转化为short类型的用户定义转换。

3.Short到int的提升。

这是个十分常见的一类转换序列：一个标准转换（一个衍生到基的转化，在此例中），接着是一个用户定义转换，然后是另一个标准转换。尽管在转换队列中最多有一个用户定义的转换，仅有一个标准转换是可能的。一个重载解析的重要原则是一个转换队列是另一个转换队列的子队列时，该队列比后面的队列更为优先。如果有一个额外的候选函数

void count(short);

在该例中，更愿意调用count(object)因为在转换序列中不需要第三步（提升）。

### C.3.3 指针转换

指针和成员指针会经历各种特殊的标准转换，包括

·转换为类型bool

·从任意指针转换为void\*

·指针的衍生到基类转换

·成员指针的基到衍生转换

尽管这些会造成“只与标准转换匹配”，它们的分级却不同。

首先，转化为类型bool（通常指针与成员指针都是）被考虑为差于其它任何类型的标准转换。例如：

void check(void\*);*// #1*

void check(bool);*// #2*

void rearrange(Matrix\* m)

{

check(m);*// 调用 #1*

}

在常规指针转换的种类中，一个转换为void\*类型被认为更差于衍生类指针到基类指针的转换。除此之外，如果转换为其他类的转换与继承关系存在，转换为最底层派生类更为优先。这里有另一个短例子：

class Interface {

};

class CommonProcesses : public Interface {

};

class Machine : public CommonProcesses {

};

char\* serialize(Interface\*);*// #1*

char\* serialize(CommonProcesses\*);*// #2*

void dump (Machine\* machine)

{

char\* buffer = serialize(machine);*// 调用 #2*

}

从Machine\*到CommonProcess的转换比Interface\*更为优先，十分直观。

一个很相似的规则适用于成员指针：在两个指针到成员类型相关的转换中，在继承图中“最接近的基类” （更少衍化）的更优先。

### C.3.4 初始化列表

初始化列表参数（initializer使用圆括号传递的）可被转换为数种不用类型的参数：initializer\_lists,带有initializer\_list的classl类型构造器，初始化列表元素数可被构造器当作参数处理的class类型，或成员可被初始化参数列表的元素初始化的聚合类。下面的例子解释类这些情况：

*overload/initlist.cpp*

*#include* <initializer\_list>

*#include* <string>

*#include* <vector>

*#include* <complex>

*#include* <iostream>

void f(std::initializer\_list<int>) {

std::cout << "#1\n";

}

void f(std::initializer\_list<std::string>) {

std::cout << "#2\n";

}

void g(std::vector<int> const& vec) {

std::cout << "#3\n";

}

void h(std::complex<double> const& cmplx) {

std::cout << "#4\n";

}

struct Point {

int x, y;

};

void i(Point const& pt) {

std::cout << "#5\n";

}

int main()

{

f({1, 2, 3});*// prints #1*

f({"hello", "initializer", "list"});*// 打印 #2*

g({1, 1, 2, 3, 5});*// 打印 #3*

h({1.5, 2.5});*// 打印 #4*

i({1, 2});*// 打印 #5*

}

在对f()的两个初次调用中，初始化列表参数被转换为std::initializer\_list值，包括将每个参数列表中的元素转换为std::initializer\_list的元素类型。在第一个调用中，所有的元素已具有int类型，故无需额外的转换。在第二次调用中，列表中的每一个string字面值通过调用string(char const\*)构造器转换为std::string。第三个调用（调用g()）执行用户定义转换，使用std::vector(std::initializer\_list<int>)构造器。下一个调用启用std::complex(double,double)构造器，就像某人写了std::complex<double>(1.5,2.5)。最后一个调用执行aggregate初始化，在未使用Point构造器的情况下使用初始化列表的元素初始化Point类的实例的成员。

有几个初始化列表中有趣的重载情况。将初始化列表转换为initializer\_list时，与上例中的前两个调用相同，所有的转换被赋予相同的等级，正如初始化列表中的任何指定元素到initializer\_list的元素类型的最差转换（initilaizer\_list<T>中的T）。这会带来一些惊奇之事，就像下例中的：

*//overload/initlistovl.cpp*

*#include* <initializer\_list>

*#include* <iostream>

void ovl(std::initializer\_list<char>) {*//#1*

std::cout << "#1\n";

}

void ovl(std::initializer\_list<int>) {*//#2*

std::cout << "#2\n";

}

int main()

{

ovl({'h', 'e', 'l', 'l', 'o', '\0'});*//打印 #1*

ovl({'h', 'e', 'l', 'l', 'o', 0});*//打印 #2*

}

在第一个对ovl()的调用中，初始化列表中每个元素都是char。对于第一个val()函数，这些元素不再需要任何转换。对于第二个ovl()函数，这些元素需要提升为int。因为最佳匹配优于提升，第一个对ovl()的调用调用#1。

在对函数ovl()的第二个调用中，前五个元素的类型为char，最后一个元素事int。对于第一个ovl()函数，char元素是最佳匹配，但int需要一个标准转换。对于第二个ovl函数，char元素需要一个到int的提升，最终int元素是一个最佳匹配。第二个ovl()函数中所有的转换被分级为一个提升，这使成为其比第一个ovl()更佳的候选者，即使只有一个元素转换也更好。

当使用初始化列表初始化一个class类型的object时，正如我们在初始例子中对g()和h()的调用时相同，重载解析执行于两个阶段：

1.第一阶段只考虑initializerl\_list构造器，也就是，对某个类型T来说构造器的唯一非默认参数的类型为std::initializer\_list<T>（在移除高层面引用和const/volatile修饰符后）。

2.若无此类可行构造器被检查到，那么第二阶段考虑所有的其它的构造器。此规则有一个例外：如果初始化列表为空且类有用默认构造器，第一阶段会被跳过所以默认构造器会被调用。

本规则的影响是任何初始化列表构造器比任何非初始化构造器更匹配，正如下例说明的：

*//overload/initlistctor.cpp*

*#include* <initializer\_list>

*#include* <string>

*#include* <iostream>

template<typename T>

struct Array {

Array(std::initializer\_list<T>) {

std::cout << "#1\n";

}

Array(unsigned n, T const&) {

std::cout << "#2\n";

}

};

void arr1(Array<int>) {

}

void arr2(Array<std::string>) {

}

int main()

{

arr1({1, 2, 3, 4, 5});*// 打印 #1*

arr1({1, 2});*// 打印 #1*

arr1({10u, 5});*// 打印 #1*

arr2({"hello", "initializer", "list"});*//打印 #1*

arr2({10, "hello"});*//打印 #2*

}

注意第二个构造器，那个接收unsigned和T const&，在从初始化列表初始化一个Array<int>对象时不会被调用，因为其初始化列表构造器总比其非初始化列表构造器更匹配。有了Array<string>，与之相反，在初始化列表构造器不可行时，非初始化列表构造器会被调用，正如对arr2()的调用。

### C.3.5 仿函数和代理函数

我们之前提到过函数名称被检查并用于创建一个初始重载集后，该集以各种方式微调。一个有趣的情景发生于调用一个关于类类型对象而非一个函数的表达式时。在这种情况下，重载集有两个重要的添加物。

加上的第一个东西很直接：任何成员操作符（）（函数调用操作符）加入到集中。拥有此类操作符的函数被称为仿函数或函数对象（157页，11.1节）。

一个更不明显的添加物在一个含有隐式转换为指针或函数类型（或函数类型引用）的类类对象中出现。在这些情况中，一个虚构（替代）函数加入到重载集中。该替代函数候选者被认为拥有一个转化版本指定的该类型的隐式参数，加上与该转换函数的目标类型的参数类型匹配的参数。一个例子可以说明的更清楚：

using FuncType = void (double, int);

class IndirectFunctor {

public:

void operator()(double, double) const;

operator FuncType\*() const;

};

void activate(IndirectFunctor const& funcObj)

{

funcObj(3, 5);*// 错误: 歧义*

}

调用funcObj(3,5)被当作具有三参的调用处理：funcObj，3和5。可行函数候选者包括了成员操作符()（被当作具有成员类型IndirectFunctor cosnt&，double，和double参数）和具有类型为FuncType\*，double，int参数的替代函数，替代函数在隐式参数上匹配更差（因为需要用户定义转换），但在最后一个参数上匹配更佳；因此两个候选者无法决定顺序。调用也因此歧义。

替代函数位于C++最隐晦的角落并且在事件中极少发生（这儿是幸运的）。

### C.3.6 其它重载情况

目前为止我们已经在在一个调用表达式中决定哪个函数应被调用的环境中讨论了重载。然而，也有少数其它情景中需要做出相似选择。

第一个情景出现在需要一个函数的地址时。考虑以下例子：

int numElems(Matrix const&);*// #1*

int numElems(Vector const&);*// #2*

...

int (\*funcPtr)(Vector const&) = numElems;*// 选择 #2*

这里，名称numElems关一个重载集，但该集中只有一个函数的地址是被需要的。之后重载解析尝试在可选候选者中匹配需要的函数类型（本例中是funcPtr类型）。

另一个需要重载解析的情景是初始化。不幸的是，这是一个充满着多于我们在一个附录中可以描述的的子主题的主题。然而，一个简单的例子至少可以阐释重载解析的这个额外层面：

*#include* <string>

class BigNum {

public:

BigNum(long n);*// #1*

BigNum(double n);*// #2*

BigNum(std::string const&);*// #3*

...

operator double();*// #4*

operator long();*// #5*

...

};

void initDemo()

{

BigNum bn1(100103);*// 选择 #1*

BigNum bn2("7057103224.095764");*//选择 #3*

int in = bn1;*// 选择 #5*

}

在本例中，重载解析需要被用于选择合适的构造器或转换操作符。明确意义上，bn1的初始化调用第一个构造器，bn2调用第三个构造器，in()调用operator long()。在大多数主要情形中重载规则产生直观结果。然而，这些规则的细节十分复杂，并且某些应用依赖于C++语言范围内那些更为隐晦的角落。

# 附录D

# 标准类型实用程序

C++标准库大部分由模板组成，这些模板中很多都依赖于在此书中介绍并讨论的各种技术。因此，标准库使用原型代码定义了数个模板来实现库，从而让人感觉到一些技术被标准化了。这些类型工具（type traits和其它帮助）在本章中列出并解释。

注意某些类型特征（type traits）需要编译器支持，其他的只需要使用库中存在的语言内置特性就可实现（我们在Chapter19讨论了一部分）。

## D.1 使用type traits

使用类型特征时，通常你需要include头文件：<type\_traits>:

*#include* <type\_traits>

之后的使用取决于是一个特征产出一个类型或一个值：

·对于产出一个类型的特征，你可以使用以下方式获取类型：

typename std::trait<…>::type

std::trait\_t<…>*// 自从 C++14 之后*

·对于产出一个值的特征，你可以使用以下方式获取值：

std::trait<…>::value

std::trait<…>()*//隐式转换为其类型*

std::trait\_v<…>*//自从 C++17 之后*

例如：

*//utils/traits1.cpp*

*#include* <type\_traits>

*#include* <iostream>

int main()

{

int i = 42;

std::add\_const<int>::type c = i;*// c 是 int const*

std::add\_const\_t<int> c14 = i;*// 自从 C++14*

static\_assert(std::is\_const<decltype(c)>::value, "c should be const");

std::cout << std::boolalpha;

std::cout << std::is\_same < decltype(c), int const>::value

*//true*

<<'\n';

std::cout << std::is\_same\_v <decltype(c),  int const>

*//自从 C++17后*

<< '\n';

*if* (std::is\_same<decltype(c), int const>{}) {*//隐式转换为bool*

std::cout << "same \n";

}

}

\_t版本特征的定义方式位于2.8节。\_v版本特征的定义方式位于5.6节。

### D.1.1 std::integral\_constant 和 std::bool\_constant

所有产生一个值的标准类型特征衍生于帮助类模板std::integral\_constant的实例：

namespace std {

template<typename T, T val>

struct integral\_constant {

static constexpr T value = val;*// 该特征的值*

using value\_type = T;*// 值的类型*

using type = integral\_constant<T,val>;

constexpr operator value\_type() const noexcept {

*return* value;

}

constexpr value\_type operator() () const noexcept {*//自从 C++14*

*return* value;

}

};

}

也就是：

·我们可以使用value\_type成员查询结果的类型。因为很多特征生成的值可预测，value\_constant通常为类型bool。

·特征类型的对象有一个隐式类型转换，转换为类型特征提供的值的类型。

·在C++14(以及之后)，类型特征的对象也是函数对象（伪函数），“函数调用”会产出他们的值。

·成员type只会生成底层的integral\_constant实例。

如果trait生成布尔值，它们也可使用。

namespace std {

template<bool B>

using bool\_constant = integral\_constant<bool, B>;*// 自从 C++17*

using true\_type = bool\_constant<true>;

using false\_type = bool\_constant<false>;

}

如果一个确切的属性被使用并且不是来自于std::false\_type，这些布尔特征继承于std::true\_type。这也意味着它们对应的value成员等价于ture或false。对于结果值为true和false拥有明显的类型允许我们进行基于特征类型的结果进行标签调度（相关章节为19.3.3节和20.2节）。

例如：

*//utils/traits2.cpp*

*#include* <type\_traits>

*#include* <iostream>

int main()

{

using namespace std;

cout << boolalpha;

using MyType = int;

cout << is\_const<MyType>::value << '\n';*//打印 false*

using VT = is\_const<MyType>::value\_type;*// bool*

using T = is\_const<MyType>::type;*//integral\_constant<bool, false>;*

cout << is\_same<VT,bool>::value << '\n';*//打印 true*

cout << is\_same<T, integral\_constant<bool, false>>::value

<<'\n';*//打印 true*

cout << is\_same<T, bool\_constant<false>>::value<<'\n';*//打印 true (not valid*

*// prior to C++17)*

auto ic = is\_const<MyType>();*// 特征类型的对象*

cout << is\_same<decltype(ic), is\_const<int>>::value << '\n';

*// true*

cout << ic() << '\n';*//函数调用 (打印 false)*

static constexpr auto mytypeIsConst = is\_const<MyType>{};

*if* constexpr(mytypeIsConst) {*//自从 C++17开始编译时检查 => false…*

...*//弃用的声明*

}

static\_assert(!std::is\_const<MyType>{}, "MyType should not be const");

}

对于非布尔integral\_constant特化来说，拥有明显的类型也在多种元编程环境中也很实用。查看相似类型CTValue的讨论，24.3节和其获取元组中元素的使用，25.6节。

### D.1.2 使用Traits时应知道的事

有一些事应在使用特征时应该注意：

·类型特征直接使用于类型，但decltype也允许我们测试表达式、变量和函数的属性。回想一下，然而，只有在实体被没有括号的名称命名时，decltype才会产生变量和函数的类型；对于其他任何表达式，它产生一个类型时也反映了表达式的类型类别。例如：

void foo (std::string&& s)

{

*// check the type of s:*

std::is\_lvalue\_reference<decltype(s)>::value*//false*

std::is\_rvalue\_reference<decltype(s)>::value*//true,正如声明*

*// 检查所用表达式的值类别:*

std::is\_lvalue\_reference<decltype((s))>::value*//true, s作为左值使用*

std::is\_rvalue\_reference<decltype((s))>::value*//false*

}

查看15.10.2节以了解细节。

·对于某些初级程序员来说某些特征有不直接的行为。相关例子位于11.2.1节。

·某些某些特征有需求或者前提。忽视这些前提会导致未定义行为。 11.2.1节有一些例子。

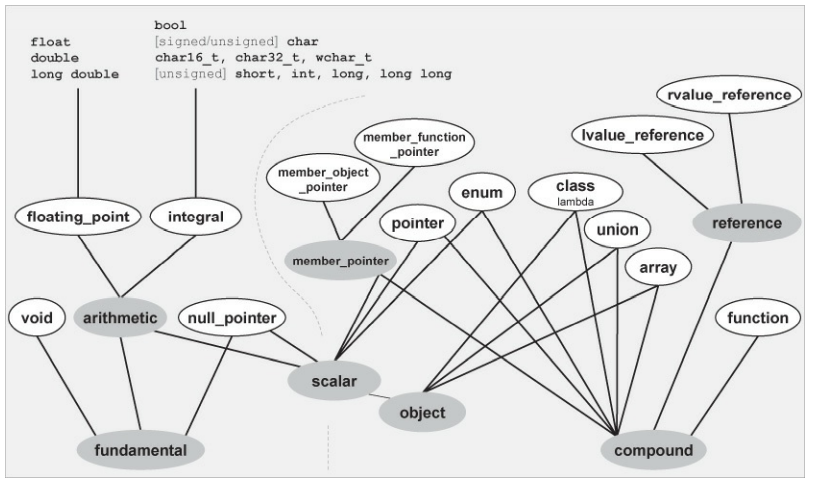
·很多特征需要完整的类型（154页，10.3.1节）。为了在不完整类型中能使用它们，我们有时可以导入模板来推导它们的赋值（细节位于171页，11.5节）。

·基于其它类型特征，有时逻辑运算符&&，||，和!无法用来基于其他类型定义新类型。此外，处理可能有故障的类型时会成为一个问题或至少产生一些倒退。基于此理由，特殊特征在被提供时允许我们逻辑性地结合Boolean特征。细节位于D.6节。

·尽管标准别名模板（以\_t 或 \_v结尾）通常很方便，它们也有消极面，这使得它们在某些元编程环境中无法使用。细节位于19.7.3节，446页。

## D.2 主要和复合类型

我们以测试主要和复合类型类别地标准特征开始（D.1的图片）。通常来说，每个类型属于准确的一个主要类型类别（图D.1中的白色元素）。复合类型范畴后来并入主要类型类别并成为高层次概念。

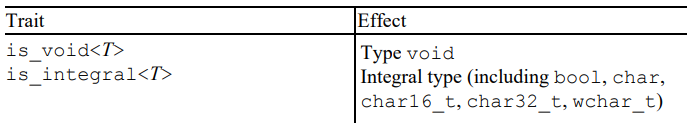


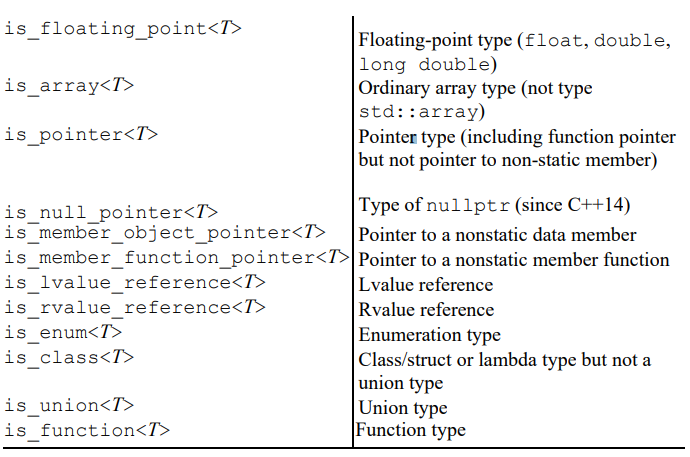
图D.1主要和复合类型类别

### D.2.1 测试主要类型的类别

本节描述了测试一个给定类型的主要类型类别使用的类型工具。对于任何给定类型，准确的主要类型类别之一有一个被赋值为true的静态value成员。该结果和类型是否被const和/或volatile修饰无关（cv-qualified）。

需要注意，对于类型std::size\_t和std::ptrdiff\_t，is\_integral<>产生true。对于类型std::max\_align\_t，这些产生true的主要类型类别之一，是一个实现细节（因此，它可以是整数或浮点数或类类型）。语言明确指出lambda表达式的类型为class类型（15.10.6节）。对该类型使用is\_class产生true。





表D.1 检查主要类型类别的特征

std:: **is\_void** < T >::value

·如果类型T为void（cv-qualified）产生true。

·例如：

is\_void\_v<void>*// 产生 true*

is\_void\_v<void const>*// 产生 true*

is\_void\_v<int>*// 产生 false*

void f();

is\_void\_v<decltype(f)>*// 产生 false (f有函数类型)*

is\_void\_v<decltype(f())>*// 产生 true (f()的返回值类型为void)*

std:: **is\_integral** < T >::value

·如果T是下列类型之一（cv-qualified），产生true：

-布尔值

-一个字符类型（char,signed char,unsigned char,char16\_t,char32\_t,或wchar\_t）

-一个整数类型（有符号或无符号的各种各样的short,int,long,或long long；这包括std::size\_t 和 std::ptrdiff\_t）

std:: **is\_floating\_point** < T >::value

·如果类型是float,double或long double（cv-qualified）产生true。

std:: **is\_array** < T >::value

·如果T是数组类型（cv-qualified），产生true。

·回想一下一个声明为数组的parameter（有或无长度）在语言规则中实际上有一个pointer类型。

·注意类 std::array<> 不是数组类型，是类类型。

·例如：

is\_array\_v<int[]>*// 产生 true*

is\_array\_v<int[5]>*// 产生 true*

is\_array\_v<int\*>*// 产生 false*

void foo(int a[], int b[5], int\* c)

{

is\_array\_v<decltype(a)>*// 产生 false (a 有一个 int\* 类型)*

is\_array\_v<decltype(b)>*// 产生 false (b 有一个 int\* 类型)*

is\_array\_v<decltype(c)>*// 产生 false (c 有一个 int\* 类型)*

}

·查看19.8.2节的实现细节。

std:: **is\_pointer** < T >::value

·如果T是一个指针（cv-qualified），产生true

这包括：

-静态/全局（成员）函数指针

-声明为数组（有或无长度）或函数类型的参数

这不包括：

-成员指针类型（例如，&X::m的类型，X是一个类类型并且m是一个非静态成员函数或非静态数据成员）

-类型为nullprt,std::nullptr\_t

例如：

is\_pointer\_v<int>*// 产生 false*

is\_pointer\_v<int\*>*// true*

is\_pointer\_v<int\* const>*//  true*

is\_pointer\_v<int\*&>*//  false*

is\_pointer\_v<decltype(nullptr)>*// false*

int\* foo(int a[5], void(f)())

{

is\_pointer\_v<decltype(a)>*// true (a 有 int\* 类型)*

is\_pointer\_v<decltype(f)>*// true (f 有 void(\*)() 类型)*

is\_pointer\_v<decltype(foo)>*// false*

is\_pointer\_v<decltype(&foo)>*// true*

is\_pointer\_v<decltype(foo(a,f))>*// true (因为返回值为 int\*)*

}

·实现细节位于19.8.2节，451页

std:: **is\_null\_pointer** < T >::value

·如果类型T是std::nullptr\_t（cv-qualified），即nullptr的类型，产生true。

·例如：

is\_null\_pointer\_v<decltype(nullptr)>*// 产生 true*

void\* p = nullptr;

is\_null\_pointer\_v<decltype(p)>*// 产生 false (p 无类型 std::nullptr\_t)*

·前提为C++14环境

std:: **is\_member\_object\_pointer** < T >::value

std:: **is\_ member\_function \_pointer** < T >::value

·如果T是一个成员指针类型（cv-qualified）（例如int X::\* 或 某个类类型的int(X::\*)），产生true

std:: **is\_lvalue\_reference** < T >::value

std:: **is\_rvalue\_reference** < T >::value

·如果类型T为左值或右值引用类型（cv-qualified）之一，生成true。

·例如：

is\_lvalue\_reference\_v<int>*// false*

is\_lvalue\_reference\_v<int&>*// true*

is\_lvalue\_reference\_v<int&&>*// false*

is\_lvalue\_reference\_v<void>*// false*

is\_rvalue\_reference\_v<int>*// false*

is\_rvalue\_reference\_v<int&>*// false*

is\_rvalue\_reference\_v<int&&>*// true*

is\_rvalue\_reference\_v<void>*// false*

·实现细节位于19.8.2节，452页

std:: **is\_enum** < T >::value

·如果类型T为枚举类型（cv-qualified），产生true。对有域和无域枚举都适用。

·实现细节位于19.8.5节。

std:: **is\_class** < T >::value

·如果T是以class或struct声明的类类型（cv-qualified），包括类模板生成实例的类类型，产生true。注意语言保证lambda表达式的类型是类类型（15.10.6节）。

·对于联合，域枚举类型（无论声明中是否带有enum class），std::nullptr\_t，以及其他任何类型生成false。

·例如：

is\_class\_v<int>*//  false*

is\_class\_v<std::string>*//  true*

is\_class\_v<std::string const>*//  true*

is\_class\_v<std::string&>*// false*

auto l1 = []{};

is\_class\_v<decltype(l1)>*// true (lambda 是类类型)*

·实现细节位于19.8.4节，456页

std:: **is\_union**< T >::value

·如果T是union（cv-qualified），包括是类模板的联合模板生成的联合，生成true.

std:: **is\_function**< T >::value

·如果T是函数类型（cv-qualified），生成true。如果是函数指针、lambda表达式以及其他类型，生成false。

·回忆一下在语言规则中，一个声明为函数类型的parameter实际上有一个指针类型。

·例如：

void foo(void(f)())

{

is\_function\_v<decltype(f)>*// false (f 有一个类型 void(\*)())*

is\_function\_v<decltype(foo)>*// true*

is\_function\_v<decltype(&foo)>*// false*

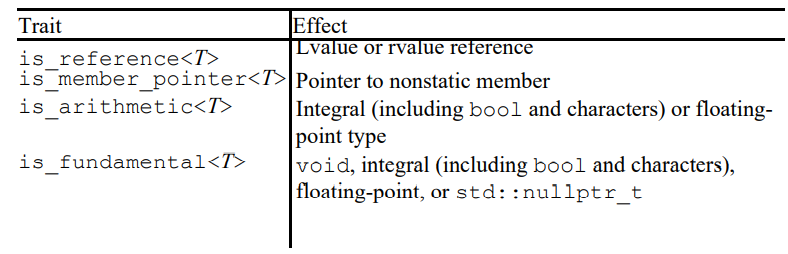
is\_function\_v<decltype(foo(f))>*// false (对于返回值类型)*

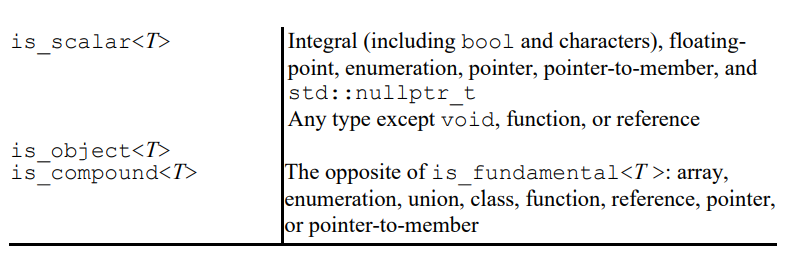
}

·实现细节位于19.8.3节，454页

### D.2.2 测试复合类型的类别

下面的类型工具决定了一个类型是否属于在某些主要的类型类别的联合中更常规的类型类别。复合类型类别并未形成严格划分：一个类型可能属于多个复合类型类别（例如，一个指针类型同时是scalar类型和compound类型）。再说一次，cv-qualifiers（const 和 vlatile）在分类时无关紧要。





表D.2 用于检查复合类型类别的特征

std:: **is\_reference**< T >::value

·如果T是引用类型，生成true。

·等价于：is\_lvalue\_reference\_v<T>|| is\_rvalue\_reference\_v<T>

·实现细节位于19.8.2节。

std:: **is\_member\_pointer**< T >::value

·如果T是任何成员指针类型，生成true。

·等价于：!(is\_member\_object\_pointer\_v<T> || is\_member\_function\_pointer\_v<T>)

std:: **is\_arithmetic**< T >::value

·如果T是一个算数类型（bool，字符类型，整数类型或浮点数类型），生成true。

·等价于: is\_integral\_v<T> || is\_floating\_point\_v<T>

std:: **is\_fundamental**< T >::value

·如果T是基础类型（算术类型，void或std::nullptr\_t）。

·等价于is\_arithmetic\_v<T> || is\_void\_v<T> || is\_null\_pointer\_v<T>·等价于! is\_compound\_v<T>·实现细节位于19.8.1节的IsFundaT。

std:: **is\_scalar**< T >::value

·如果T是“scalar”类型，生成true。

·等价于：is\_arithmetic\_v<T> || is\_enum\_v<T> || is\_pointer\_v<T> || is\_member\_pointer\_v<T> || is\_null\_pointer\_v<T>>

std:: **is\_object**< T >::value

·如果类型T描述了一个对象的类型，生成true。

·等价于：(is\_scalar\_v <T>|| is\_array\_v<T> || is\_class\_v<T> || is\_union\_v<T>)

·等价于：! (is\_function\_v || is\_reference\_v || is\_void\_v)

std:: **is\_compound**< T >::value

·如果类型T是其他类型的组成的复合类型，生成true。

·等价于：!is\_fundamental\_v<T>

·等价于：is\_enum\_v<T> || is\_array\_v<T> || is\_class\_v<T> ||

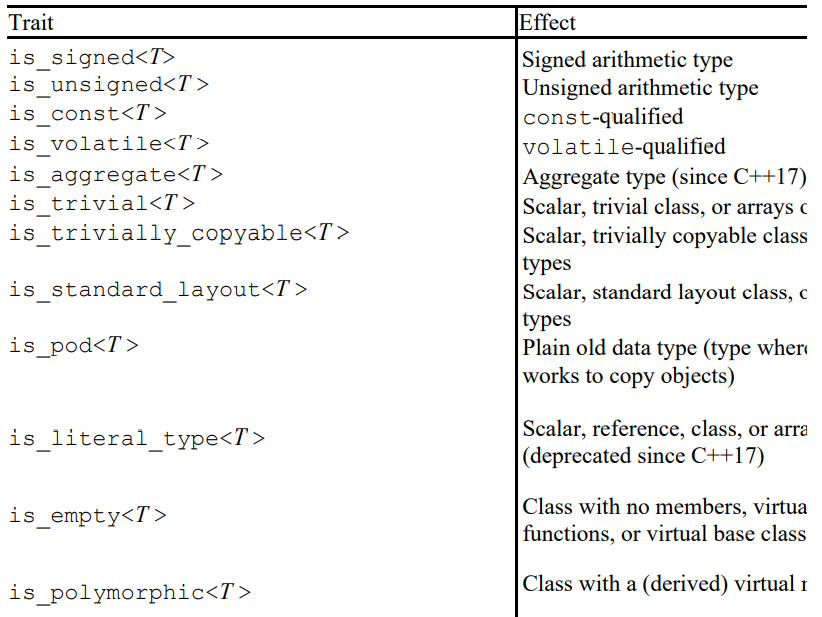
is\_union\_v<T>|| is\_reference\_v<T> || is\_pointer\_v<T> || is\_member\_pointer\_v<T>

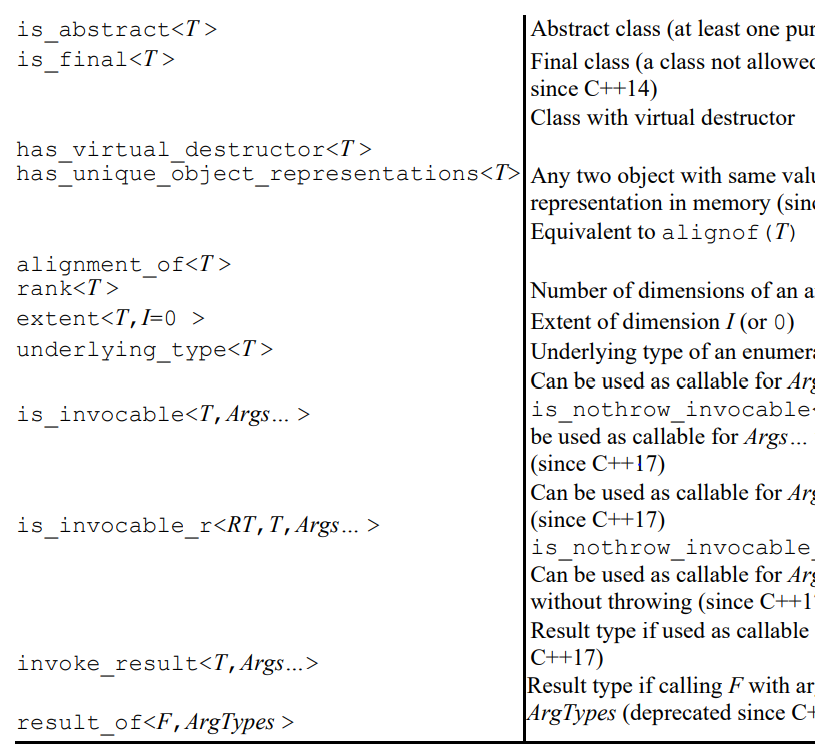
|| is\_function\_v<T>

## D.3 类型属性和操作

下面一组特征，正如可能应用于他们的确切操作一样，测试单个类型的其他属性（例如，交换值）。

### D.3.1 其它类型属性





表D.3 测试简单类型属性的特征

std:: **is\_signed**< T >::value

·如果类型T是有符号算术类型（例如，一个包括负值表示的算术类型；这包括类似于（signed）int，float），生成true。

·对于类型bool，产生false。

·对于类型char，其实现定义了它是否产生true或false。

·对于所有非算术类型（包括枚举类型）is\_signed产生false。

std:: **is\_unsigned**< T >::value

·如果T是一个无符号算术类型（例如，一个不包括负值表示的算术类型；这包括的类型有unsigned int和bool），产生true。

·对于类型char，其实现定义了它是否产生true或false。

·对于所有非算术类型（包括枚举类型）is\_unsigned产生false。

std:: **is\_const**< T >::value

·如果类型使用const修饰，产生true。

·注意一个const指针有一个const修饰类型，同时一个指向const类型的非const指针或引用不是const修饰的，例如：

is\_const<int\* const>::value*// true*

is\_const<int const\*>::value*// false*

is\_const<int const&>::value*// fals*

·语言规定如果数组中的元素类型是const修饰的，则数组也是const修饰的。例如：

is\_const<int[3]>::value*// false*

is\_const<int const[3]>::value*// true*

is\_const<int[]>::value*// false*

is\_const<int const[]>::value*// true*

std:: **is\_volatile**< T >::value

·如果类型T被volatile修饰，生成true。

·注意一个volatile指针有一个volatile类型，而指向volatile类型的非volatile指针或引用并不是volatile修饰。例如：

is\_volatile<int\* volatile>::value*// true*

is\_volatile<int volatile\*>::value*// false*

is\_volatile<int volatile&>::value*// false*

·语言定义了如果数组的元素是volatile修饰，数组也是volatile修饰。

例如：

is\_volatile<int[3]>::value*// false*

is\_volatile<int volatile[3]>::value*// true*

is\_volatile<int[]>::value*// false*

is\_volatile<int volatile[]>::value*// true*

std:: **is\_aggregate**< T >::value

·如果T是一个aggregate类型（任意一个没有用户定义的、显式的、或继承的构造器，没有private或protected非静态数据成员，没有虚函数，没有virtual、private或protected基类的数组或类/结构体/联合），生成true。

·用于指明是否需要一个初始化列表。例如：

template<typename Coll, typename… T>

void insert(Coll& coll, T&&… val)

{

*if* constexpr(!std::is\_aggregate\_v<typenameColl::value\_type>) {

coll.emplace\_back(std::forward<T>(val)…);*// 对于聚合无效*

}

*else* {

coll.emplace\_back(typename Coll::value\_type{std::forward<T>

(val)…});

}

}

·需要给定类型的一个为完整的（10.3.1节）或void（cv-qualified）。

·自C++17后可用

std:: **is\_trival**< T >::value

·如果是“trival”类型，生成true：

-一个scalar类型（integral,float,enum,pointer；is\_scalar<>位于707页）

-一个通常的类类型（一个没有虚函数，虚基类，没有（间接）用户定义默认构造器，copy/move构造器，copy/move赋值操作符，或析构器，没有非静态数据成员的初始化器，没有volatile成员，和没有非通常成员）。

-该类型的数组。

-这些类型的cv-qualified版本。

·如果is\_trivially\_copyable\_v<T>产生ture并且常规的默认构造函数存在，产生true。

·需要给定类型的一个为完整的（10.3.1节）或void（cv-qualified）。

std:: **is\_trivally\_copyable**< T >::value

·如果是“trivally copyable”类型，生成true。

-一个scalar类型（integral,float,enum,pointer；is\_scalar<>位于707页）

-一个通常的类类型（没有虚函数，虚基类，（间接）用户定义默认构造器，copy/move构造器，copy/move赋值操作符，或析构器，没有非静态数据成员的初始化器，没有volatile成员，和没有非通常成员）

-该类型的数组。

-这些类型的cv-qualified版本。

·除了可以对没有通常的默认构造器的类产生true之外，和is\_trivial\_v<T>的生成相同。

·和is\_standard\_layout<>相反，volatile成员不被允许，引用被允许，成员可能有不同的获取方式，成员可能会被分布于不同的（基）类。

·需要给定类型的一个为完整的（10.3.1节）或void（cv-qualified）。

std:: **is\_standard\_layout**< T >::value

·如果类型拥有标准布局，例如，使其更容易的与其他语言中相同类型互换值，产生true。

-一个scalar类型（integral,float,enum,pointer；is\_scalar<>位于707页）

-一个standard-layout类型（没有虚函数，虚基类，没有非静态引用成员，所有非静态成员在相同的（基）类中并定义了相同的获取方式，所有的成员也都是standard-layout类型）。

-该类型的数组。

-这些类型的cv-qualified版本。

·和is\_trival<>相反，volatile成员被允许，引用不被允许，成员可能不能有不同的获取，成员可能不会被分布于不同的（基）类。

·需要给定的类型（对于数组，基础类型）之一为完整的（10.3.1节）或void（cv-qualified）。

std:: **is\_pod**< T >::value

·如果T是一个plain old datatype(POD)，产生true。

·该类型的对象可以通过复制底层存储进行复制（例如，适用memcpy（））。

·等价于：is\_trivial\_t<T> && is\_standard\_layout\_v<T>·对以下产生false：

-没有常规默认构造器，copy/move构造器，copy/move赋值构造器，或析构器的class。

-有虚函数或虚基类的类

-有volatile或reference的成员

-有位于不同（基）类成员或不同的成员获取方式的类

-lambda表达式类型（称为closuer type）

-函数

-void

-这些类型复合的类型

·需要给定的类型之一为完整的（10.3.1节）或void（cv-qualified）。

std:: **is\_literal\_type**< T >::value

·如果给定类型对constexpr函数是一个有效的返回值类型，产生true（明显的去除任何需要非常规析构的类型）。

·如果T是literal type，产生true:

-一个scalar类型（integral,float,enum,pointer；is\_scalar<>位于707页）

-一个引用

-至少有一个不在每个（基）类中的copy/move构造器的constexpr构造器、在任何（基）类或成员中没有用户定义或虚析构器、并且每个非静态数据成员的初始化是一个常量表达式。

·此类型的任何数组

·需要给定的类型之一为完整的（10.3.1节）或void（cv-qualified）。

·在C++17中被弃用该特征已被弃用，因为“在源码中有意义的使用它太过无力。真正需要的是知道一个确定的构造器可能会产生constant初始化的能力。”

std:: **is\_empty**< T >::value

·如果T是类类型但不是union类型，其对象保存数据，生成true。

·如果T定义为class或struct时有以下内容，生成true:

-没有非静态数据成员，除了长度为0的bit-fields

-没有虚成员函数

-没有虚基类

-没有非空基类

·如果类型为class/struct（不完整的union也可以）需要给定的类型是完整的（10.3.1节）。

std:: **is\_ploymorphic**< T >::value

·如果T是多态类类型（一个声明或继承虚函数的类），产生true。

·需要给定类型之一是完整的（10.3.1节）或既不是class也不是struct。

std:: **is\_abstract**< T >::value

·如果T是abstract类类型（没有对象能生成的类，因为它有至少一个纯虚成员函数），生成true。

·如果它是class/struct（不完整的union也可以），需要给定的类型是完整的（10.3.1节）。

std:: **is\_final**< T >::value

·如果T是final类类型（不能作为基类的类或联合因为被声明为final），生成ture。

·对于所有非类/联合类型，例如int，他返回false（因此，这不同于类似is derivable的属性）。

·需要给定类型T之一为完整的（10.3.1节）或既不是class/struct也非union。

·自C++14后可用

std:: **has\_virtual\_destructor**< T >::value

·如果T有一个虚析构器，生成true。

·如果它是class/struct（不完整的union也可以），需要给定的类型是完整的（10.3.1节）。

std:: **has\_unique\_object\_representations**< T >::value

·两个类型T的对象在内存中有相同的对象表示，生成true。也就是，两个不同的值总是以相同的字节值表示。

·拥有此类属性的对象可以哈希化相关字节序列后（某些字节不参与对象值可能在不同的情况下不同的危险不存在）产生一个可靠的哈希值。

·前提是提供的类型易于复制（D.3.1节）并且是完整的（10.3.1节）或(cv\_qualified)void或大小未定的数组。

·自C++17后可用

std:: **alignment\_of**< T >::value

·产生一个作为std::size\_t的类型T的对象的对齐值（对于数组，元素的类型；对于引用，被引用的类型）。

·等价于alignof(T)。

·在C++11中引入该特性引入，早于alignof()构造器。它仍然有用，因为该特征可以在类类型之间被传递，这特定的元编程中很实用。

·前提是alignof(T)是一个有效的表达式。

·使用aligned\_union<>来获得多种类型的通常对齐（D.5节）。

std:: **rank**< T >::value

·产生作为std::size\_t的类型T的数组的维度数。

·对所有其他类型生成0。

·指针没有任何相关的维度。一个未指明边界的数组类型并未指出一个维度（和往常相同，以数组类型声明的函数参数时并无数组类型，并且std::array也不是一个数组类型。D.2.1节）。

例如：

int a2[5][7];

rank\_v<decltype(a2)>;*//产生 2*

rank\_v<int \*>;*//产生 0(不是数组)*

extern int pl[];

rank\_v<decltype(p1)>;*//产生 1s*

std:: **extent**< T >::value

std:: **exent**< T,IDX >::value

·产生作为std::size\_t的类型T的数组的第一个或IDX-th维度。

·如果T不是数组，维度数不存在或维度的大小未知，产生0。

·实现细节位于19.8.2节。

int a2[5][7];

extent\_v<decltype(a2)>;*// 产生 5*

extent\_v<decltype(a2),0>;*// 产生 5*

extent\_v<decltype(a2),1>;*// 产生 7*

extent\_v<decltype(a2),2>;*// 产生 0*

extent\_v<int\*>;*// 产生 0*

extern int p1[];

extent\_v<decltype(p1)>;*// 产生 0*

std::**underlying\_type**< T >::value

·产生枚举类型T的底层类型。

·需要给定的类型是完整的枚举类型（10.3.1节）。对于所有其他类型，它有未定义行为。

std::**is\_invocable**< T, Args …>::value

std::**is\_nothrow\_invocable**< T, Args… >::value

·如果T对Args可调用（需要保证不会抛出异常），产生true。

·也就是，我们可以使用这些特征来测试，使用Args…，我们是否可以调用或std::invoke()给定的callable T（有关callable和std::invoke（）的细节，查看11.1节）。

·前提是给有定的类型是完整的（10.3.1节）或（cv-qualified）void或一个未知边界的数组。

·例如：

struct C {

bool operator() (int) const {

*return* true;

}

};

std::is\_invocable<C>::value*//false*

std::is\_invocable<C,int>::value*//true*

std::is\_invocable<int\*>::value*//false*

std::is\_invocable<int(\*)()>::value*//true*

·自C++17后可用

std::**is\_invocable\_r**<RET\_T,T, Args …>::value

std::**result\_of**< RET\_T,T, Args …>::value

·如果T对Args可调用（需要保证不会抛出异常），返回一个可转换为类型RET\_T的值。

·也就是，我们可以使用这些特征来测试，使用Args…，我们是否可以调用或std::invoke()给定的callable T并使用返回值作为RET\_T。（有关callable和std::invoke（）的细节，查看11.1节）。

·前提是给有定的类型是完整的（10.3.1节）或（cv-qualified）void或一个未知边界的数组。

例如：

std::is\_invocable\_r<bool,C,int>::value*//true*

std::is\_invocable\_r<int,C,long>::value*//true*

std::is\_invocable\_r<void,C,int>::value*//true*

std::is\_invocable\_r<char\*,C,int>::value*//false*

std::is\_invocable\_r<long,int(\*)(int)>::value*//false*

std::is\_invocable\_r<long,int(\*)(int),int>::value*//true*

std::is\_invocable\_r<long,int(\*)(int),double>::value*//true*

·自C++17后可用

std::**invoke\_result**< T, Args …>::value

std::**result\_of**< T, Args …>::value

·对与Args产生callable T的返回值。

·注意分析过程有一点不同：

-对于invoke\_result<>，你必须同时传递可调用的类型和变量的类型作为参数。

-对于result\_of<>，你必须使用对应的类型传递一个“函数声明”。

·如果不调用是可行的，那没有type成员定义，使用它就是错误的（可能使用其本身的定义SIFNE出一个函数模板；查阅8.4节）。

·也就是说，对于Args，当我们调用或std::invoke()给定的callable T，我们可以使用这些特性获取获得的返回类型（有关callable和std::invoke（）的细节，查看11.1节）。

·前提是所有给定的类型是完整的（10.3.1节）或（cv-qualified）void或一个未知边界的数组。

·invoke\_result<>在C++17后可用并替换result\_of<>，它也在C++17中被弃用。因为invoke\_result<>提供了一些优化，例如对T的更早的分析和允许抽象类型。

例如：

std::string foo(int);

using R0 = typename std::result\_of<decltype(&foo)

(int)>::type;*// C++11*

using R1 = std::result\_of\_t<decltype(&foo)(int)>;*// C++14*

using R2 = std::invoke\_result\_t<decltype(foo), int>;*// C++17*

struct ABC {

virtual ~ABC() = 0;

void operator() (int) const {

}

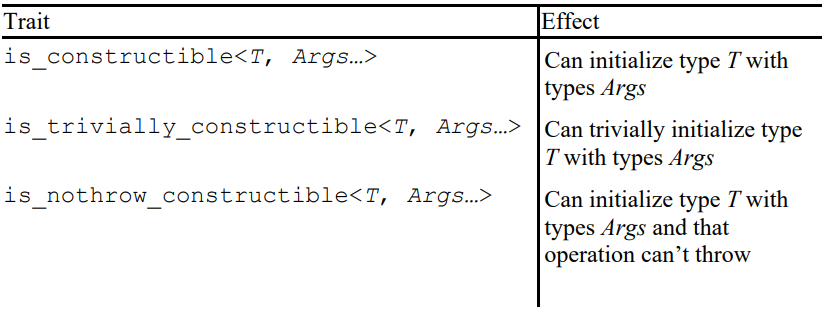
};

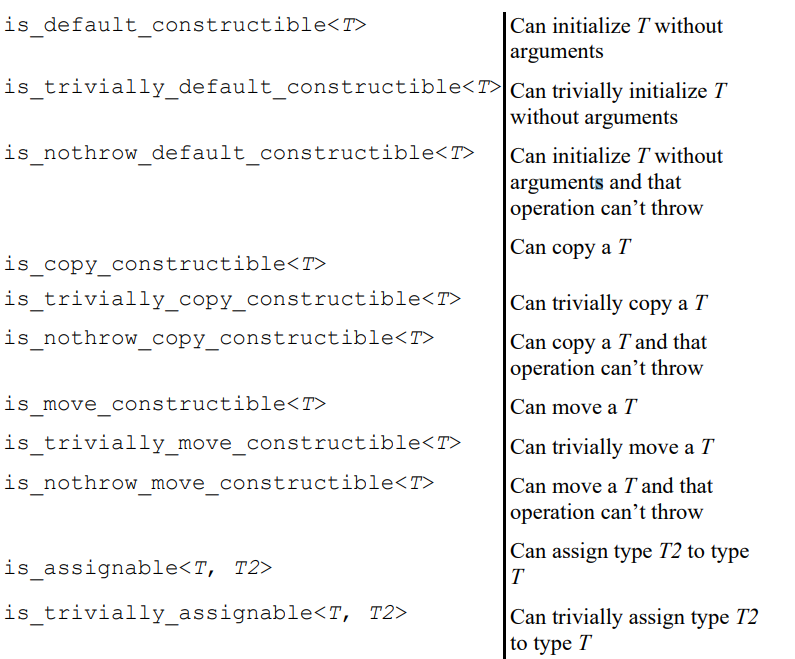
using T1 = typename std::result\_of<ABC(int)>::type;*// ERROR: ABC 是抽象的*

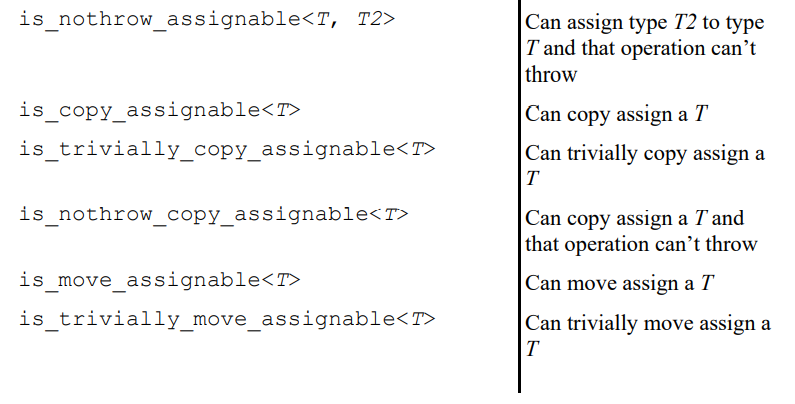
using T2 = typename std::invoke\_result<ABC, int>::type;*// OK 自从 C++17*

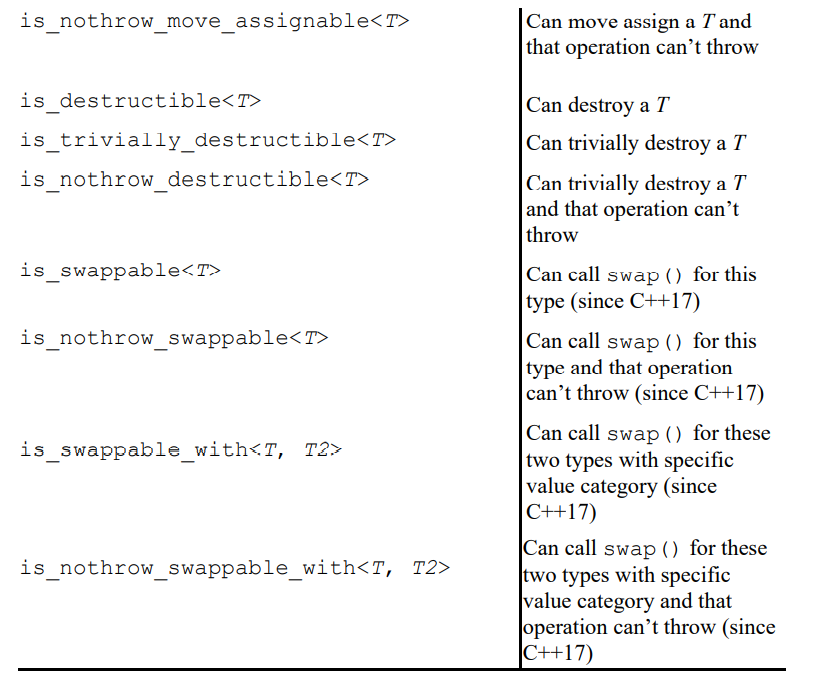
查阅11.1.3节，163页获得全例。

### D.3.2 测试具体操作









表D.4 检查特定操作的特征

表D.4列出了允许我们检查一些特定操作的类型特征。带有is\_trivially\_...形式的可以额外检查对象、成员或基类的所有（子）调用是否是通常的（无论是用户定义的或虚的）。带有is\_nothrow\_...额外检查调用的操作是否保证不被抛出。注意所有的is\_...\_constructive检查暗含这对应的is\_...\_destructible检查。例如：

*//utils/isconstructible.cpp*

*#include* <iostream>

class C {

public:

C() {*//没有noexcept的默认构造器*

}

virtual ~C() = default;*// 使得 C nontrivial*

};

int main()

{

using namespace std;

cout << is\_default\_constructible\_v<C> << ’\n’;*//true*

cout << is\_trivially\_default\_constructible\_v<C> << ’\n’;*//false*

cout << is\_nothrow\_default\_constructible\_v<C> << ’\n’;*//false*

cout << is\_copy\_constructible\_v<C> << ’\n’;*//true*

cout << is\_trivially\_copy\_constructible\_v<C> << ’\n’;*//true*

cout << is\_nothrow\_copy\_constructible\_v<C> << ’\n’;*//true*

cout << is\_destructible\_v<C> << ’\n’;*//true*

cout << is\_trivially\_destructible\_v<C> << ’\n’;*//false*

cout << is\_nothrow\_destructible\_v<C> << ’\n’;*//true*

}

因virtual构造器的定义，所有的操作不再是通常的。并且因为我们定义一个默认构造器时加上了noexcept，他可能抛出异常。其他所有操作，默认情况下不会抛出异常。

std::**is\_constructible**< T, Args …>::value

std::**is\_trivially\_constructible**< T, Args …>::value

std::**is\_nothrow\_constructible**< T, Args …>::value

·如果一个类型为T的对象可以被具有Args给定类型的参数初始化，生成true（不使用nontrivial操作或带有不抛出异常的保证）。也就是下面的表达式应该有效：

T t(std::declval<Args>()…);

·一个true值意味着该对象可以按照一定方式销毁（例如，is\_destructiible\_v<T>，is\_trivially\_destructible\_v<T>，或is\_nothrow\_destructible\_v<T>生成true）。

·前提是给定的类型是完整的（10.3.1节）或（cv-qualified）void或一个未知边界的数组。

例如：

is\_constructible\_v<int>*// true*

is\_constructible\_v<int,int>*// true*

is\_constructible\_v<long,int>*// true*

is\_constructible\_v<int,void\*>*// false*

is\_constructible\_v<void\*,int>*// false*

is\_constructible\_v<char const\*,std::string>*// false*

is\_constructible\_v<std::string,char const\*>*// true*

is\_constructible\_v<std::string,char const\*,int,int>*// true*

·注意is\_convertible对于源和目的类型有不同的顺序。

std::**is\_default\_constructible**< T>::value

std::**is\_trivially\_default\_constructible**< T>::value

std::**is\_nothrow\_ default\_constructible**< T>::value

·如果一个类型为T的对象可以被不给定任何类型的参数初始化，生成true（不使用nontrivial操作或带有不抛出异常的保证）。

·等价于：std::**is\_constructible\_v**< T>

std::**is\_trivially\_constructible\_v**< T>

std::**is\_nothrow\_constructible\_v**< T>

一一对应。

·一个true值意味着该对象可以按照一定方式销毁（例如，is\_destructiible\_v<T>，is\_trivially\_destructible\_v<T>，或is\_nothrow\_destructible\_v<T>生成true）。

·前提是给定的类型是完整的（10.3.1节，154页）或（cv-qualified）void或一个未知边界的数组。

std::**is\_copy\_constructible**< T>::value

std::**is\_trivially\_ copy\_constructible**< T>::value

std::**is\_nothrow\_ copy\_constructible**< T>::value

·如果T可以通过复制另一个类型T的值来被创建，生成true（不使用nontrivial操作或带有不抛出异常的保证）。

·如果T不是一个可以引用类型（（cv-qualified）void 或 被const,volatile,和/或&&修饰的函数），生成false。

·前提是T是一个可引用对象，各自等价于is\_constructible<T, T const&>::value, is\_trivially\_constructible<T, T const&>::value，is\_nothrow\_constructible<T , T const&>::value

·为查明一个T对象能否被一个类型T的右值复制构造，使用is\_constructible<T , T &&>，等等。

·true值表示该对象可以按序销毁（例如，is\_destructible\_v<T>，is\_trivially\_destructible\_v<T>，或is\_nothrow\_destructible\_v<T>生成true）

·前提是给定的类型是完整的（10.3.1节，154页）或（cv-qualified）void或一个未知边界的数组。

例如：

is\_copy\_constructible\_v<int>*// 生成 true*

is\_copy\_constructible\_v<void>*// false*

is\_copy\_constructible\_v<std::unique\_ptr<int>>*//  false*

is\_copy\_constructible\_v<std::string>*// true*

is\_copy\_constructible\_v<std::string&>*// true*

is\_copy\_constructible\_v<std::string&&>*// false*

*// 与之相反：*

is\_constructible\_v<std::string,std::string>*// true*

is\_constructible\_v<std::string&,std::string&>*// true*

is\_constructible\_v<std::string&&,std::string&&>*// true*

std::**is\_move\_constructible**< T>::value

std::**is\_trivially\_ move\_ constructible** < T>::value

std::**is\_nothrow\_ move\_ constructible** < T>::value

·如果一个类型为T的对象可以通过另一个类型为T的右值创建，生成true（不使用nontrivial操作或带有不抛出异常的保证）。

·如果T不是一个可以引用类型（（cv-qualified）void 或 被const,volatile,和/或&&修饰的函数），生成false。

·前提是T是一个可引用对象，各自等价于is\_constructible<T, T &&>::value, is\_trivially\_constructible<T, T &&>::value，is\_nothrow\_constructible<T , T&&>::value

·true值表示该对象可以按序销毁（例如，is\_destructiible\_v<T>，is\_trivially\_destructible\_v<T>，或is\_nothrow\_destructible\_v<T>生成true）

·在不能对类型T的对象直接调用的情况下，没有方法检查一个move构造器是否抛出异常抛出异常。让构造器是public且不能删除是不够的；它也需要对应类型不是一个抽象类（抽象类的引用或指针效果不错）。

·实现细节位于19.7.2节。

例如：

is\_move\_constructible\_v<int>*// true*

is\_move\_constructible\_v<void>*// false*

is\_move\_constructible\_v<std::unique\_ptr<int>>*// true*

is\_move\_constructible\_v<std::string>*// true*

is\_move\_constructible\_v<std::string&>*// true*

is\_move\_constructible\_v<std::string&&>*// true*

*// 与之相反*

is\_constructible\_v<std::string,std::string>*// true*

is\_constructible\_v<std::string&,std::string&>*// true*

is\_constructible\_v<std::string&&,std::string&&>*// true*

std::**is\_assignable**< TO, FROM>::value

std::**is\_trivially\_ assignable** < TO, FROM >::value

std::**is\_nothrow\_ assignable** < TO, FROM >::value

·如果一个类型为FROM的对象可以为类型为TO的对象赋值（不使用nontrivial操作或带有不抛出异常的保证），生成true。

·需要给定的类型是完整的（10.3.1节）或（cv-qualified）void或一个未知边界的数组。

·注意is\_assignable\_v<>对于非引用，非类类型作为第一个类型时只返回false，因为此类型生成纯右值。也就是，表达式42=77；是无效的。然而对于类类型右值可能被赋值，给定一个合适的赋值操作符（因为一个non-const成员函数可被类类型的右值调用的旧规则）。

·注意is\_convertible对于源和目的类型有不同的顺序。

例如：

is\_assignable\_v<int,int>*// false*

is\_assignable\_v<int&,int>*// true*

is\_assignable\_v<int&&,int>*// false*

is\_assignable\_v<int&,int&>*// true*

is\_assignable\_v<int&&,int&&>*// false*

is\_assignable\_v<int&,long&>*// true*

is\_assignable\_v<int&,void\*>*// false*

is\_assignable\_v<void\*,int>*// false*

is\_assignable\_v<void\*,int&>*// false*

is\_assignable\_v<std::string,std::string>*// true*

is\_assignable\_v<std::string&,std::string&>*// true*

is\_assignable\_v<std::string&&,std::string&&>*// true*

std::**is\_copy\_assignable**< T>::value

std::**is\_trivially \_copy\_ assignable** < T>::value

std::**is\_nothrow\_ copy\_ assignable** < T >::value

·如果一个类型为T的值可以给类型为T的对象copy赋值（不使用nontrivial操作或带有不抛出异常的保证），生成true。

·如果T不是一个可以引用类型（（cv-qualified）void 或 被const,volatile,和/或&&修饰的函数），生成false。

• 前提是 T 是可引用类型, 各自等价于 is\_assignable<T&,T const&>::value, is\_trivially\_assignable<T&,T const&>::value, 或 is\_nothrow\_assignable<T&,T const&>::value。

·为找出一个类型为T的右值能否被另一个类型为T的右值copy赋值，使用is\_assignable<T&&,T&&>,等等。

·注意void,内置数组类型和删除了copy赋值操作符的类无法被copy赋值。

·前提是给定的类型是完整的（10.3.1节，154页）或（cv-qualified）void或一个未知边界的数组。

例如：

is\_copy\_assignable\_v<int>*// true*

is\_copy\_assignable\_v<int&>*// true*

is\_copy\_assignable\_v<int&&>*// true*

is\_copy\_assignable\_v<void>*// false*

is\_copy\_assignable\_v<void\*>*// true*

is\_copy\_assignable\_v<char[]>*// false*

is\_copy\_assignable\_v<std::string>*// true*

is\_copy\_assignable\_v<std::unique\_ptr<int>>*// false*

std::**is\_move\_assignable**< T>::value

std::**is\_trivially \_move\_ assignable** < T >::value

std::**is\_nothrow\_ move\_ assignable** < T >::value

·如果一个类型为T的右值可以给类型为T的对象move赋值（不使用nontrivial操作或带有不抛出异常的保证），生成true。

·如果T不是一个可以引用类型（（cv-qualified）void 或 被const,volatile,和/或&&修饰的函数），生成false。

• 前提是 T 是可引用类型, 各自等价于 is\_assignable<T&,T &&>::value, is\_trivially\_assignable<T&,T &&>::value, 或 is\_nothrow\_assignable<T&,T &&>::value。

·注意void,内置数组类型和删除了move赋值操作符的类无法被move赋值。

·前提是给定的类型是完整的（10.3.1节，154页）或（cv-qualified）void或一个未知边界的数组。

·例如：

is\_move\_assignable\_v<int>*// true*

is\_move\_assignable\_v<int&>*// true*

is\_move\_assignable\_v<int&&>*// true*

is\_move\_assignable\_v<void>*// false*

is\_move\_assignable\_v<void\*>*// true*

is\_move\_assignable\_v<char[]>*// false*

is\_move\_assignable\_v<std::string>*// true*

is\_move\_assignable\_v<std::unique\_ptr<int>>*// true*

std::**is\_destructible**< T>::value

std::**is\_trivially\_ destructible** < T>::value

std::**is\_nothrow\_ destructible** < T>::value

·如果类型为T的对象可被摧毁，生成true（不使用nontrivial操作或带有不抛出异常的保证）。

·对于引用只生成true。

·对于void，未知边界的数组类型，函数类型生成false。

·std::is\_trivially\_ destructible < T>对于没有T自身的、任何基类或任何用户定义或虚静态数据成员的析构器时，生成true。

·前提是给定的类型是完整的（10.3.1节，154页）或（cv-qualified）void或一个未知边界的数组。

·例如：

is\_destructible\_v<void>*//生成 false*

is\_destructible\_v<int>*// true*

is\_destructible\_v<std::string>*// true*

is\_destructible\_v<std::pair<int,std::string>>*// true*

is\_trivially\_destructible\_v<void>*// false*

is\_trivially\_destructible\_v<int>*// true*

is\_trivially\_destructible\_v<std::string>*// false*

is\_trivially\_destructible\_v<std::pair<int,int>>*// true*

is\_trivially\_destructible\_v<std::pair<int,std::string>>*// false*

std::**is\_swappable\_with**< T1 , T2 >::value

std::**is\_nothrow\_ swappable\_with** < T1 , T2 >::value

·除了引用只决定表达式的值类别外（保证无异常抛出），如果类型T1的表达式可以与类型T2的表达式交换值，生成true。

·前提是给定的类型是完整的（10.3.1节）或（cv-qualified）void或一个未知边界的数组。

·注意is\_swappable\_with\_v<>作为第一或第二个类型的非引用、非类产生false，因为此类生成纯右值。也就是，swap(42,77)无效。

例如：

is\_swappable\_with\_v<int,int>*// 生成 false*

is\_swappable\_with\_v<int&,int>*// false*

is\_swappable\_with\_v<int&&,int>*// false*

is\_swappable\_with\_v<int&,int&>*// true*

is\_swappable\_with\_v<int&&,int&&>*// false*

is\_swappable\_with\_v<int&,long&>*// false*

is\_swappable\_with\_v<int&,void\*>*// false*

is\_swappable\_with\_v<void\*,int>*// false*

is\_swappable\_with\_v<void\*,int&>*// false*

is\_swappable\_with\_v<std::string,std::string>*// false*

is\_swappable\_with\_v<std::string&,std::string&>*// true*

is\_swappable\_with\_v<std::string&&,std::string&&>*// false*

·C++17后可用

std::**is\_swappable**< T >::value

std::**is\_nothrow\_ swappable** < T >::value

·如果类型T的左值可以被交换，生成true（保证无异常抛出）。

·前提是T是一个引用类型。各自等价于std::is\_swappable\_with< T& , T& >::value或

std::is\_nothrow\_ swappable\_with < T& , T& >::value。

·若想找出一个T的右值能否与另一个T的右值互换，使用is\_swappable\_with<T&&,T&&>。

·前提是给定的类型是完整的（10.3.1节，154页）或（cv-qualified）void或一个未知边界的数组。

例如：

is\_swappable\_v<int>*// 生成 true*

is\_swappable\_v<int&>*// true*

is\_swappable\_v<int&&>*// true*

is\_swappable\_v<std::string&&>*// true*

is\_swappable\_v<void>*// false*

is\_swappable\_v<void\*>*// true*

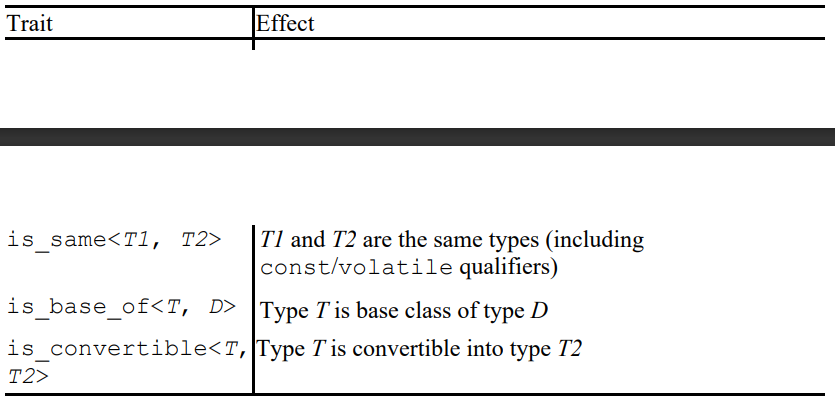
is\_swappable\_v<char[]>*// false*

is\_swappable\_v<std::unique\_ptr<int>>*// true*

·自C++17后可用

### D.3.3 类型间的关系

表D.5列出了允许测试类型间特定关系的类型特征。这包括检查为类类型提供了哪个构造器和赋值操作符。



表D.5 测试类型关系的特征

std::**is\_same**< T1 , T2 >::value

·如果T1和T2命名相同的类型，包括cv\_qualifiers（const和 volatile），生成true。

·如果一个类型是另一类型的别名，生成true。

·如果两个对象以同一类型的对象初始化，生成true。

·对于（封闭的）与两个不同的lambda表达式的类型生成false，即使它们定义了相同的行为。

·例如：

auto a = nullptr;

auto b = nullptr;

is\_same\_v<decltype(a),decltype(b)>*//生成 true*

using A = int;

is\_same\_v<A,int>*// true*

auto x = [] (int) {};

auto y = x;

auto z = [] (int) {};

is\_same\_v<decltype(x),decltype(y)>*// true*

is\_same\_v<decltype(x),decltype(z)>*// false*

·实现细节位于19.3.3节，410页。

std::**is\_base\_of**< B , D >::value

·如果B是D的基类或B与D是同一类。

·一个类无论是否cv-qualified，使用private或protected继承，D有多个类型B的基类，或D通过多重继承方式继承了B（通过虚继承），都无关。

·如果至少有一个类型是union，生成true。

·前提是类型D是完整的（10.3.1节），有与B相同的类型（忽略任何const/volatile修饰），或既不是struct也不是class。

·例如：

class B {

};

class D1 : B {

};

class D2 : B {

};

class DD : private D1, private D2 {

};

is\_base\_of\_v<B, D1>*// 生成 true*

is\_base\_of\_v<B, DD>*// true*

is\_base\_of\_v<B const, DD>*// true*

is\_base\_of\_v<B, DD const>*// true*

is\_base\_of\_v<B, B const>*// true*

is\_base\_of\_v<B&, DD&>*// false (非 class 类型)*

is\_base\_of\_v<B[3], DD[3]>*// false (非 class 类型)*

is\_base\_of\_v<int, int>*// false (非 class 类型)*

std::**is\_convertible**< FROM , TO >::value

·如果类型为FROM的表达式可转换为类型TO，生成true。因此，下列的代码应当有效：

TO test() {

*return* std::declval<FROM>();

}

·类型FORM之上的引用只用于决定被转换表达式的值类别；底层类型之后才是原表达式的类型。

·注意is\_constructible**不**总是意味着is\_convertible。例如：

class C {

public:

explicit C(C const&);*// 非隐式 copy 构造器*

…

};

is\_constructible\_v<C,C>*// 生成 true*

is\_convertible\_v<C,C>*// 生成 false*

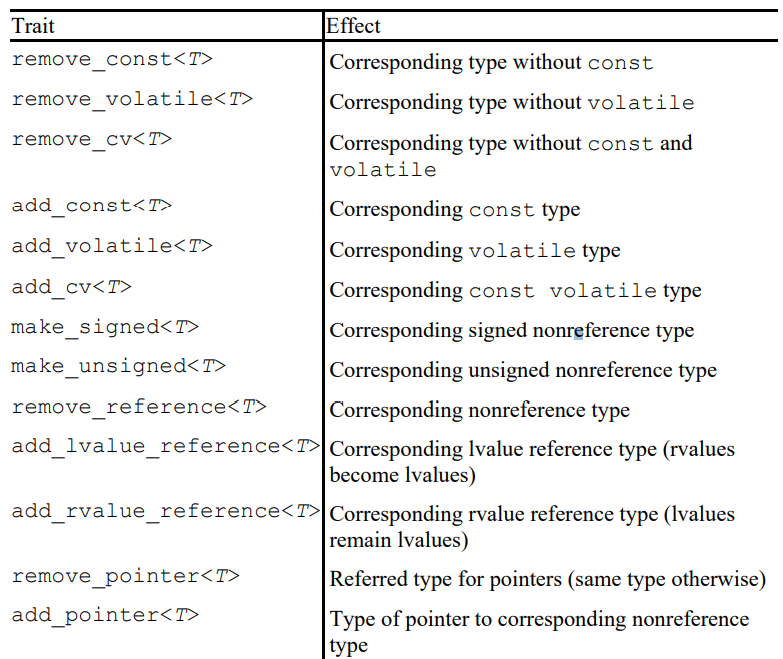
·前提是给定的类型是完整的（10.3.1节）或（cv-qualified）void或一个未知边界的数组。

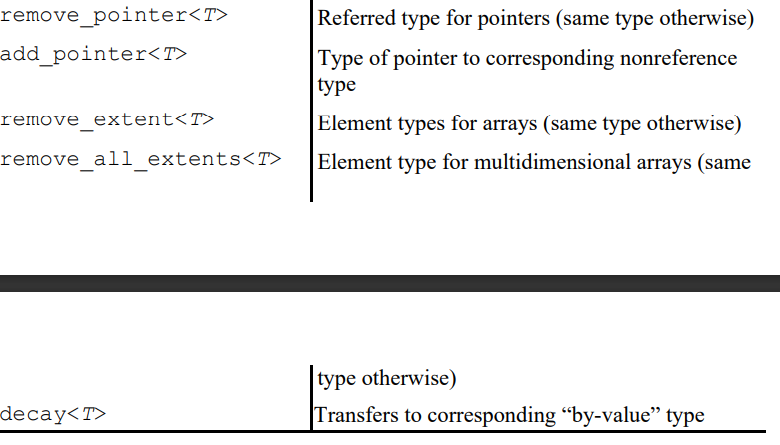
·注意对于is\_constructible(D.3.2节)和is\_assignable(D.3.2节)对于源和目的类型有不同的顺序。

·实现细节位于19.5节。

## D.4 类型结构

表D.6中列出的特性允许我们从其他类型中构建类型。





表D.6 用于类型构造的特性

std::**remove\_const**< T >::type

std::**remove\_volatile**< T >::type

std::**remove\_cv**< T >::type

·生成其上方没有const和/或volatile的类型T。

·注意const指针是const修饰的类型，而对const类型的non-const指针或引用不是const修饰。例如：

remove\_cv\_t<int>*// 生成 int*

remove\_const\_t<int const>*// int*

remove\_cv\_t<int const volatile>*// int*

remove\_const\_t<int const&>*// int const& (只引用int const)*

显而易见，类型构造特性使用的顺序有影响：

remove\_const\_t<remove\_reference\_t<int const&>>*// int*

remove\_reference\_t<remove\_const\_t<int const&>>*// int const*

取而代之，我们可能更想用std::decay<>，然而这也将数组和函数类型转换为对应的指针类型（D.4节）。

·实现细节位于19.3.2节。

std::**add\_const**< T >::type

std::**add\_volatile**< T >::type

std::**add\_cv**< T >::type

·生成T的类型时加上const和/或volatile修饰符。

·将这些特征之一用于引用类型或函数类型时无效，例如：

add\_cv\_t<int>*// int const volatile*

add\_cv\_t<int const>*// int const volatile*

add\_cv\_t<int const volatile>*// int const volatile*

add\_const\_t<int>*// int const*

add\_const\_t<int const>*// int const*

add\_const\_t<int&>*// int&*

std::**make\_signed**< T >::type

std::**make\_unsigned**< T >::type

·生成类型T对应的有符号/无符号类型。

·前提是T是一个枚举类型或除布尔类型外的整数（cv-qualified）。所有其他类型导致未定义行为（查看19.7.1节，有关如何避免此类未定义行为的讨论）。

·将此特征之一用于引用类型或函数类型无效，此外一个指向const类型的非const指针或引用并未const修饰。例如：

make\_unsigned\_t<char>*// unsigned char*

make\_unsigned\_t<int>*// unsigned int*

make\_unsigned\_t<int const&>*// 未定义行为*

std::**remove\_reference**< T >::type

·生成引用类型T所引用的类型（或者T本身不是一个引用类型）。

例如：

remove\_reference\_t<int>*// int*

remove\_reference\_t<int const>*// int const*

remove\_reference\_t<int const&>*// int const*

remove\_reference\_t<int&&>*// int*

·注意引用类型本身不是const类型。因此，应用类型构建特征的顺序会有所影响：

remove\_const\_t<remove\_reference\_t<int const&>>*// int*

remove\_reference\_t<remove\_const\_t<int const&>>*// int const*

取而代之的是，我们偏向于使用std::decay<>，也将数组和函数类型转换为对应的指针类型（D.4节）：

decay\_t<int const&>*// int*

·实现细节位于19.3.2节

std::**add\_lvalue\_reference**< T >::type

std::**add\_rvalue\_reference**< T >::type

·如果T是可以引用类型，生成T的左值或右值引用。

·如果T无法引用，生成T（无论（cv-qualified）void或被const,volatile,&,和/或&&修饰的函数）。

·注意如果T早已是一个引用类型，该特征使用引用收缩规则（15.6.1节，277页）：只有add\_rvalue\_reference被使用且T是右值引用时结果才是右值引用。

·例如

add\_lvalue\_reference\_t<int>*// int&*

add\_rvalue\_reference\_t<int>*// int&&*

add\_rvalue\_reference\_t<int const>*// int const&&*

add\_lvalue\_reference\_t<int const&>*// int const&*

add\_rvalue\_reference\_t<int const&>*// int const&*

(reference collapsing rules)

add\_rvalue\_reference\_t<remove\_reference\_t<int const&>>*//*

yields int&&

add\_lvalue\_reference\_t<void>*// void*

add\_rvalue\_reference\_t<void>*// void*

·实现细节位于19.3.2节

std::**remove\_pointer**< T >::type

·生成指针类型T指向的类型（或T不是指针类型）

·例如：

remove\_pointer\_t<int>*// int*

remove\_pointer\_t<int const\*>*// int const*

remove\_pointer\_t<int const\* const\* const>*// int const\* const*

std::**add\_pointer**< T >::type

·产生指向T的指针类型，或者在T是引用的情况下，生成指向T底层类型的指针类型。

·如果没有此类型生成T（适用于cv-qualified函数类型）。

·例如：

add\_pointer\_t<void>*// void\**

add\_pointer\_t<int const\* const>*// int const\* const\**

add\_pointer\_t<int&>*// int\**

add\_pointer\_t<int[3]>*// int(\*)[3]*

add\_pointer\_t<void(&)(int)>*// void(\*)(int)*

add\_pointer\_t<void(int)>*// void(\*)(int)*

add\_pointer\_t<void(int) const>*//  void(int) const (不变)*

std::**remove\_extent**< T >::type

std::**remove\_all\_extents**< T >::type

`·给定一个数组类型，remove\_extent产生其当前元素的类型（他们本身可能是数组类型）并且remove\_all\_extents去除所有“数组层面”来提供底层元素类型（他也因此不是数组类型）。如果T不是数组类型，会产生T本身。

·指针于维度无关。一个未指明边界的数组类型并未确定一个维度。（通常来说，声明为数组类型的函数参数实际上并没有数组类型，并且std::array也不是一个函数类型。请看D.2.1节）。

·例如：

remove\_extent\_t<int>*// int*

remove\_extent\_t<int[10]>*// int*

remove\_extent\_t<int[5][10]>*// int[10]*

remove\_extent\_t<int[][10]>*// int[10]*

remove\_extent\_t<int\*>*// int\**

remove\_all\_extents\_t<int>*// int*

remove\_all\_extents\_t<int[10]>*// int*

remove\_all\_extents\_t<int[5][10]>*// int*

remove\_all\_extents\_t<int[][10]>*// int*

remove\_all\_extents\_t<int(\*)[5]>*// int(\*)[5]*

·实现细节位于23.1.2节，531页。

std::**decay**< T >::type

·产生T的衰退类型

·细节上来说，对于类型T下面的转换会被执行：

-首先，remove\_reference（D.4节，729页）被执行。

-如果结果为数组类型，产生一个当前元素类型的指针（7.1节）。

-否则，如果结果为函数类型，产生为该函数的add\_pointer生成的类型（11.1.1节）。

-否则，产生的结果没有任何顶层的const/volatile修饰符

·在初始化一个类型为auto的对象时，decay<>模型化了参数的传值或类型转换。

·decay<>在处理可能被引用代替但用于决定返回值类型或另一个函数的参数类型的模板时很实用。查看1.3.2节和7.6节讨论的例子以及适用std::decay<>()（后者带有实现std::make\_pair<>()的历史）。

·例如：

decay\_t<int const&>*// int*

decay\_t<int const[4]>*// int const\**

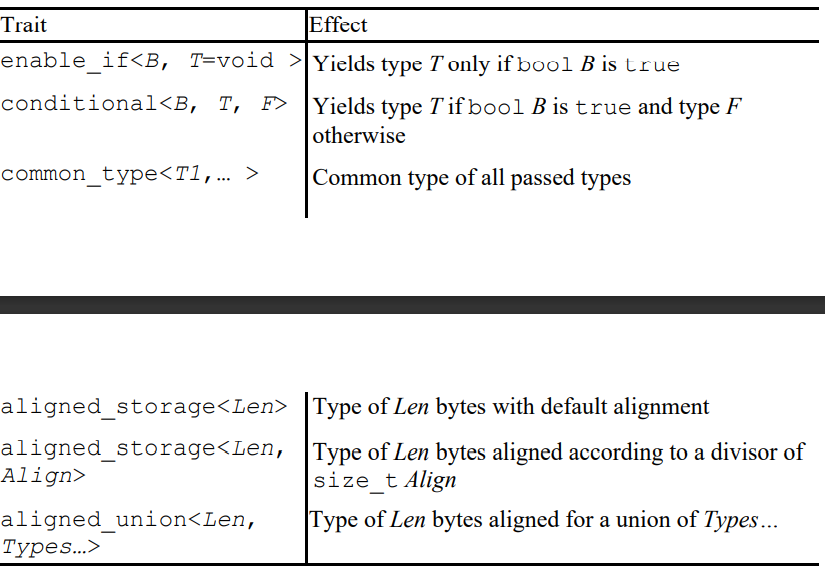
void foo();

decay\_t<decltype(foo)>*// void(\*)()*

·实现细节位于19.3.2节，407页。

## D.5 其它Trait

表D.7列举了剩余的所有类型特征。它们查询特定属性或提供更复杂的类型转换。



表D.7其它类型特征

std::**enable\_if**< cond >::type

std::**enable\_if**< cond ,T>::type

·生成void，如果cond是true，生成其成员类型之中的T。否则，它不定义一个成员类型。

·因为cond是false时type成员未被定义，该特征可以且常被用于在给定情况下无效化或SFINAE一个函数模板。

·查看6.3节获取有关第一个例子的细节。查看D.6另一个使用参数包的例子。

·查看20.3节获取std::enable\_if如何实现的相关细节。

std::**conditional**< cond, T, F >::type

·若cond为true，生成T，否则F。

·这是19.7.1节引入的特征IfThenElseT的标准版本。

·注意，不同于通常的C++ if-then-else表示，对两个分支“then”和“else”的模板参数都在被选择前被计算，所以两个分支都不会含有不规范的代码或程序看起来不规范。结果是如果该分支未被使用，你可能必须加上中间层来避免在“then”和“else”分支中该表达式被计算。19.7.1节演示了特征IfThenElseT的例子，这两者具有相同行为。

·查看11.5节获取实例

·查看19.7.1节获取std::conditional如何实现的细节。

std::**common\_type**< T… >::type

·生成给定类型T*1*，T*2*，…，T*n*的“普通类型”。

·普通类型的计算比我们想在此书中涉及的更为复杂。粗略的讲，两个类型U和V的普通类型是当第二和第三个操作数是类型U和V的时候，条件操作符:?产生的（引用类型只用于决定两个操作数的值类别）；如果它无效，则没有普通类型。Decat\_t（731页）被用于该结果。该默认计算可能通过用户确定的std::common\_type<U,V>被覆写（在C++标准库中，部分特化因持续性和时间点而存在）。

·如果没有给出类型或没有普通类型存在，也就没有type成员定义，于是使用它是一个错误（这可能使用它SFINAE一个函数模板）。

·如果被给出一个单一类型，结果是对该类型应用decay\_t的结果。

·对于两种以上的类型，common\_type根据他们的普通类型递归的替换前两个类型T1和T2。任何时候如果该过程失败，则没有通常类型。

·当处理通常类型时，传递的类型会衰退，所以该特征总会产生一个衰退类型（D.4节）。

·查看1.3.3节获取该特征的应用的讨论和例子。

·该特征的主要模板的核心通常被类似于以下的东西实现（这里只使用两个参数）：

template<typename T1, typename T2>

struct common\_type<T1,T2> {

using type = std::decay\_t<decltype(true ? std::declval<T1>()

                                        : std::declval<T2>())>;

};

std::**aligned\_union**< MIN\_SZ, T… >::type

·生成一个至少大小为MIN\_SZ并且适合容纳任何给定类型T*1*,T*2*,…，T*n*的plain old datatype(POD)作为未初始化存储的使用。

·此外，它生成具有的值与所有给定类型最严格对齐的静态成员alignment\_value，其结果类型等价于

-std::alignment\_of\_v<***type***>::value（D.3.1节）

-alignof（type）

·前提为至少提供一个类型

·例子：

using POD\_T = std::aligned\_union\_t<0, char,

                                    std::pair<std::string,std::string>>;

std::cout << sizeof(POD\_T) << ’\n’;

std::cout << std::aligned\_union<0, char,

                                std::pair<std::string,std::string>

                                >::alignment\_value;

          << ’\n’;

注意需要使用aligned\_union替代aligned\_union\_t获取对齐的值而非类型。

std::**aligned\_storage**< MAX\_TYPE\_SZ >::type

std:: **aligned\_storage**< MAX\_TYPE\_SZ ,DEF\_ALIGN >::type

·生成一个可以使用的plain old data type（POD），它拥有可以容纳MAX\_TYPE\_SZ中所有类型的未初始化的存储，默认对齐和DEF\_ALIGN传递的对齐都计入其中。

·前提是MAX\_TYPE\_SIZE大于0并且平台上至少有一种DEF\_ALIGN对齐值的类型。

·例子：

using POD\_T = std::aligned\_storage\_t<5>;

## D.6 组合Type Trait

在大多数环境中，多种类型特征论断可以通过使用逻辑操作符将其组合。然而，在某些模板元编程的环境中，这有所不足：

·如果你必须处理可能失败的特征（例如，不完整类型所致）。

·如果你想组合类型特征定义。

类型特征std::conjunction<>，std::disjunction<>，和std::negation<>供此目的使用。

例子之一是这些帮助（程序）短路布尔运算（在&&的第一个false或||的true之后中止赋值）。例如，如果使用了不完整类型：

struct X {

X(int);*// 从int转换而来*

};

struct Y;*// 不完整类型*

下面的代码可能不会编译，因为is\_constructible为不完整类型导致未定义行为（虽然，有些编译器接受这些代码）：

*// 未定义行为:*

static\_assert(std::is\_constructible<X,int>{}>

                || std::is\_constructible<Y,int>{},

            "can’t init X or Y from int");

取而代之的是，下面的代码被保证编译，因为is\_constructible<X,int>的赋值早已生成true：

*// OK:*

static\_assert(std::disjunction<std::is\_constructible<X, int>,

                std::is\_constructible<Y, int>>{},

            "can’t init X or Y from int");

另一个应用是通过逻辑上组合存在的类型特征，用一种简单的方式定义新的类型特征。比如，你可以轻松的定义一个检查一个类型是否“not a pointer”的特征（不是指针，成员指针或空指针）：

template<typename T>

struct isNoPtrT :

std::negation<std::disjunction<std::is\_null\_pointer<T>,

                                std::is\_member\_pointer<T>,

                                std::is\_pointer<T>>>

{

};

这里我们不能使用逻辑操作符，因为我们组合了对应的特征类。拥有这个定义，下面的代码是可行的：

std::cout << isNoPtrT<void\*>::value <<

'\n';*//false*

std::cout << isNoPtrT<std::string>::value <<

'\n';*//true*

auto np = nullptr;

std::cout << isNoPtrT<decltype(np)>::value <<

'\n';*//false*

如果加上对应的可变模板：

template<typename T>

constexpr bool isNoPtr = isNoPtrT<T>::value;

我们可以写出：

template<typename… Ts>

std::enable\_if\_t<std::conjunction\_v<std::negation

<std::is\_class<Ts>>…,

                    std::negation<std::is\_union<Ts>>…

                                >>

print(Ts…)

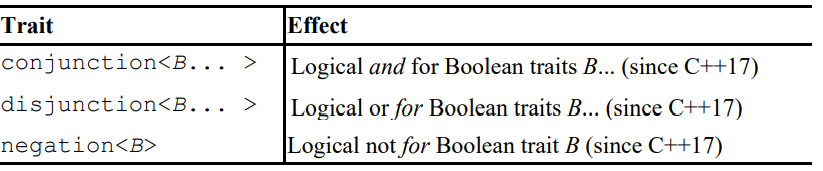
{

…

}

注意省略号被放置于每个反面之后，这样它适用于参数包的每个元素。

萃取对于布尔萃取B…（C++17之后），影响conjunction<B…>逻辑and，disjunction<B…>逻辑或（C++17之后）和negation<B>逻辑not（C++17后）。



表D.8 组合其它类型特征的类型特征

std::**conjunction**< B… >::value

std::**disjunction**< B… >::value

·生成所有的或传递的布尔特征B…之一是否生成true。

·逻辑上应用操作符&&和||，各自地传递特征。

·两个特征都是短路的（在第一个true或false后中止赋值）。看看上面启发性的例子。

·C++17后可用。

std::**negation**< B >::value

·生成传递的布尔特征B是否产生false。

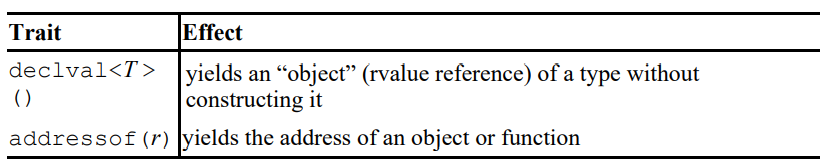
·逻辑上应用操作符！于传递地特征

·看看上面启发性地例子。

·C++17后可用。

## D.7 其他应用程序

C++标准库提供了一些其它广泛使用于写出便携源码地工具。



表D.9 元编程的其它工具

std::**declval**< T >()

·定义于头文件<utiliy>。

·不调用任何构造器或初始化，生成一个任何类型的“object”或函数。

·如果T是void，返回类型为void

·可用于任何具有未被计算表达式类型的函数或对象

·它简单的定义如下：

template<typename T>

add\_rvalue\_reference\_t<T> declval() noexcept;

因此：

-如果T是一个朴素类型或者右值引用，生成一个T&&。

-如果T是一个左值引用，它生成T&。

-如果T是void，生成void。

·查看19.3.4节的有关细节以及11.2.3节和D.5节的common\_type<>类型特征的用例。

std::**addressof**(r)::value

·定义于头文件<memory>

·生成函数或对象r的地址，即使该类型的operator&被重载。

·细节位于11.2.2。

# 附录E

**概念**

多年至今，C++语言设计者已经探索了如何约束模板中的参数。例如，在我们原型性的max()模板中，我们喜欢在前面声明它不应该被不能使用小于号比较的类型。其他类型可能需要被有效“迭代器”类型（对该术语的某些形式化定义）或有效“算数”类型（对于内置算术类型来说更为宽泛的记法）的实例化。

一个concept是在一个或多个模板上已被命名的一系列约束。当C++已被发展壮大时，为其设计了一个十分丰富的concept系统，但讲特性集成入语言的细化最终会需要太多委员会的资源，于是该版本的concepts最终从C++11中抛弃。一段时间后，该特性的一种不同的设计被提出，并且很明显的最终将要以某种形式融入语言。事实上，在本书将要出版时，标准委员会决议将该新设计整合到C++20的草案中。在此附录中，我们描述该较新设计的主要元素。

我们已经在书中主要章节启发并展示了一些concepts的使用：

·6.5节解释了如何使用requirements和concepts来使能一个构造器，只要模板参数可转换为一个string（为避免意外将构造器作为赋值构造器使用）。

·18.4节展示了如何使用concepts确定并获取用于表示几何对象类型的约束。

## E.1 使用概念

我们首先检查如何在客户代码（例如，定义模板时没有必要的用于模板参数的concepts的定义）上使用concepts。

**处理需求**

这是我们习惯性上的两参max模板，带有一个约束：

template<typename T>requires LessThanComparable<T>

T max(T a,T b){

*return* b < a ? a:b;

}

只加上了require从句。

requires LessThanComparable<T>

它假定我们已提前声明—很有可能通过一个头文件包含—concept LessThanComparable。

该concept是一个布尔断言（例如，一个产生类型为bool的值的表达式），对常量表达式估值。这很重要，因为该约束在编译时刻被估值并因此在生成源码方面的不超越：该约束模板将会提供和我们别处讨论的一样快的代码。

当我们尝试使用此模板，直到require从句被衡量并提供一个true值才会被初始化。如果它产生一个 false值，一个error可能被抛出用来解释哪个需要的部分失败了（或者，一个（不会不）满足需要的匹配的重载模板可能会被选中）。

需要从句不一定要以concepts的方式表示（即使这么做是好的实践并会产生更好的诊断）：可以使用任何布尔常量表达式。例如，正如6.5节讨论的，下面的代码保证了模板构造器不能作为复制模板构造器使用：

class Person

{

private:

std::string name;

public:

template<typename STR>

requires std::is\_convertible\_v<STR,std::string>

explicit Person(STR&& n)

: name(std::forward<STR>(n)) {

std::cout << "TMPL-CONSTR for ’" << name << "’\n";

}

…

};

这里，不使用命名的concept（E.2节）是合适的，因为特别的布尔值表达式（再此例中使用类型萃取）

std::is\_convertible\_v<STR,std::string>

被用于修复模板构造器可能被当作复制构造器使用的问题。如何组织concepts和约束的细节仍然是C++社区探索的一个乐土，并且会随着时间发展，但有concept应该反应代码的意思并于编译位置无关的一般协定。

**处理多种需求**

在上面的例子中，只有一个需求，但有多个需求不是不寻常的。例如，有人可能想象一个描述了一系列元素值（在标准上匹配相同的符号）的Sequence概念和一个模板find（），给定一个序列和一个值，返回指向在该队列中第一次出现该元素的迭代器（如果有），之类的sequence。该模板可能定义如下：

template<typename Seq>

requires Sequence<Seq> &&

EqualityComparable<typename Seq::value\_type>

typename Seq::iterator find(Seq const& seq,

typename Seq::value\_type const& val)

{

*return* std::find(seq.begin(), seq.end(), val);

}

这里，任何对此模板的调用首先按序检查每个需求，只有所有需求生成true，模板才能被该调用选中且实例化（假定，也是理所应当的，重载解析并不因其它原因丢弃模板，比如另一个模板更加匹配）。

使用||表达“可选的”前提也是可能的。这种操作的需求很少而且不能太随意的完成，因为在需求从句中过多的使用操作符||可能潜在的对编译器资源课税（例如，使得编译可见性的变慢）。然而，在某些情况下很方便。例如：

template<typename T>

requires Integral<T> ||

FloatingPoint<T>

T power(T b, T p);

单一前提也能参与多模板参数，并且单一concept可以表达在多个模板参数中表达一个断言。例如：

template<typename T, typename U>

requires SomeConcept<T, U>

auto f(T x, U y) -> decltype(x+y)

因此，concepts可以强制实行类型参数间的关系。

**单一前提的快速记法**

为在需求从句中点亮符号性，当一个约束只参与一个参数时，一个分析型的速记法是可以获取的。这可能对我们上面受约束的max()模板的声明中使用速记法来最为简单的解释：

template<LessThanComparable T>

T max(T a, T b) {

*return* b < a ? a : b;

}

这与前面的max()的定义功能性等价。当重定义一个被约束的模板时，不同的是，必须采用与初始模板相同的形式（从这方面来说它是功能等价的，但不是等价的）。

我们可以对拥有两个需求的find()模板使用相同的速记法：

template<Sequence Seq>

requires EqualityComparable<typename Seq::value\_type>

typename Seq::iterator find(Seq const& seq,

typename Seq::value\_type const& val)

{

*return* std::find(seq.begin(), seq.end(), val);

}

再次强调，这对队列类型来说和前一find()模板的定义等价。

## E.2 定义概念

Concepts更像bool类型的constexpr变量模板，但类型未明确指出：

template<typename T> concept LessThanComparable = … ;

这里“…”可能被使用多种萃取来建立类型T是否需要真正使用<操作符进行比较的表达式替换，但concpets提议提供一个工具来简化此项工作：requires 表达式（与上面描述的requires 从句完全不同）。下面是该concepts的完整定义的可能看起来的样子：

template<typename T>

concept LessThanComparable = requires(T x, T y) {

{ x < y } -> bool;

};

注意需求表达式能如何包含一个可选参数列表：这些参数从不会被变量替换，但可以被认为是一系列的“模型变量”用于表达需求表达式体中的前提。在此情况下，只有一个此类的前提在语句中表达：

{ x < y } -> bool;

该分析意味着（a）表达式x < y必须在SFINAE情境下有效并且（b）该表达式的结果必须能转换为bool。在此形式的语句中，关键字noexcept可以在->符号之前插入来表达该括号中的表达式应该被告知不能抛出异常。（例如，用于该表达式的noexcept(…)应为true）。如果没有此类约束该语句的隐式转换（->*type*）可以一同留下，并且只有表达式的有效性被检查后，括号才能丢弃，于是语句简化为一个表达式。例如：

template<typename T>

concept Swappable = requires(T x, T y) {

swap(x, y);

};

需求表达式也能表达对相关类型的需要。考虑我们前面假设的Sequence concept:为额外的查询类似于seq.begin()的表达式的有效性，它也需要对应的队列迭代器类型，可以用以下方式来表达：

template<typename Seq>

concept Sequence = requires(Seq seq) {

typename Seq::iterator;

{ seq.begin() } -> Seq::iterator;

…

};

所以语句tyname *type*；表述了要求了*type*的存在（这被称为type requirement）。在此例中，必须存在的类型是concept模板参数成员，但这也不总是情况如此。我们应该，举个例子，要求存在一个类型ItertatorFor(Seq)来替换，于是可以使用requirement-phase完成：

…

typename IteratorFor<Seq>;

…

上面列举的Sequence concept展示了如何将语句使用一句一句列举的方式结合。需求语句的有第三类，只由调用另一个concept构成。例如，假设我们对于符号iterator有一个concept。我们想让我们的Sequence concept不仅要求Seq::iterator是个类型，而且要求该类型满足Iterator concept的约束。在下面表达出来：

template<typename Seq>

concept Sequence = requires(Seq seq) {

typename Seq::iterator;

requires Iterator<typename Seq::iterator>;

{ seq.begin() } -> Seq::iterator;

…

};

也就是，我只需要在需求语句后加上需求从句（这种语句成为nested requirement）。

## E.3 重载约束

稍微设想一下，如果我们定义了concepts IntegerLike<T>和StringLike<T>，并且我们决定写一个模板来打印每个concept中每个类型的值。我们可以用以下方式处理：

template<IntegerLike T> void print(T val);*// #1*

template<StringLike T> void print(T val);*// #2*

如果没有推断约束，这两个声明可能声明了相同的模板。然而，约束只是模板签名的一部分并允许模板在重载解析时可分辨。特殊情况下，如果如果两个模板被判断为可行候选者，但只有模板#1满足其约束，重载会选择满足的条件达模板。例如：假设Int满足IntegerLike且std::string满足StringLike，但不是反之亦然：

int main()

{

printf(1);*// 选择模板 #1*

printf("1"s);*//选择模板 #2*

}

我们可以想象一个类字符串且支持类整数的计算。例如，如果”6”\_NS和”7”\_NS是该类型的两个字面值，乘这些字面值可以产生与”42”\_NS相同的值。该类型可能同时满足IntegerLike和StringLike并且，因此，一个类似于print(“42”\_NS)的调用可能是歧义的。

### E.3.1 约束包含

我们第一个对重载函数模板的讨论是在约束中引入期望下通常可以互相排斥的约束来区分它们。例如，我们IntegerLike和StringLike的例子，我们可以展望两个concept都满足的类型，但我们希望我们重载过的print模板依然有用是常常不足的。

事实是，与之相反，一系列的concept从不互相排斥而是包含另一个。一个经典的例子是标准库的迭代器类别：输入迭代器，前进迭代器，双向迭代器，随机获取迭代器，以及，C++17中，临近迭代器。假设我们有一个对ForwardIterator的定义：

template<typename T>

concept ForwardIterator = …;

那么“更为精要”的concept BidirectionalIterator的定义可能如下：

template<typename T>

concept BidirectionIterator =

ForwardIterator<T> &&

requires (T it) {

{ --it } -> T&

};

也就是，我们加入了前进迭代器早已提供的、在顶端应用前缀operator--能力。

现在思考一下std::advance()算法（我们会称他为advanceIter()），使用已约束模板重载为前进和双向迭代器：

template<ForwardIterator T, typename D>

void advanceIter(T& it, D n)

{

assert(n >= 0);

*for* (; n != 0; --n) { ++it; }

}

template<BidirectionalIterator T, typename D>

void advanceIter(T& it, D n)

{

*if* (n > 0) {

*for* (; n != 0; --n) { ++it; }

} *else* *if* (n < 0) {

*for* (; n != 0; ++n) { --it; }

}

}

当使用朴素前进迭代器调用advanceIter()时（例如，一个不是双向迭代器的迭代器），只有第一个模板的的约束会被满足，重载解析很直接：选择第一个模板。然而，一个双向迭代器会满足两个模板的约束。在此例中就像这样，当重载解析没有比它另外更佳候选者时，它会倾向包含了（subsume）其它候选者约束的候选者，反之则不同。被包含者的精确定义有点超出了这个引导性的附录，但足以让我们知道如果C2<...>的定义中需要一个约束C1<Ts...>和额外的约束（例如，&&），那么前者包括了后者。很明显，在我们的例子中，BidirectionalIterator<T>包含了ForwardIterator，因此第二个advanceIter()模板在调用具有双向迭代器时更优。

### E.3.2 约束与标签调度

回忆一下，20.2节，我们指出了使用tag dispatching重载advanceIter()算法的事件。该方法可以以十分优雅的方式被嵌入到约束模板中。例如，输入迭代器和前进迭代器在他们的分析层面不可分辨。所以，我们可以借用tags从一个方面定义另一个：

template<typename T>

concept ForwardIterator =

InputIterator<T> &&

requires {

typename std::iterator\_traits<T>::iterator\_category;

is\_convertible\_v<std::iterator\_traits<T>::iterator\_category,

std::forward\_iterator\_tag>;

};

有了它，ForwardIterator<T>包含了InputIterator<T>，并且我们现在可以对两个迭代器类别重载模板约束。

## E.4 概念提示

尽管C++ concepts至今位置运作了许多年，并且实验性的实现在十年内以某种形式可以获取，对于它们的广泛经验才刚开始涌现。我们希望本书的未来版本能够提供更多如何设计被约束模板库的实践性引导。同时，我们提供三个发现。

### E.4.1 测试概念

Concepts是布尔逻辑断言，在常量表达式中有效。因此，给定concept C和一些类型T1，T2，...模型化concept，我们可以静态断言该发现：

static\_assert(C<T1, T2, …>, "Model failure");

当设计concepts时，因此推荐也涉及一个简单类型以这种方式来测试它们。这包括的能将其推向concept能接受边界的类型，回答如下问题：

·接口和/或算法需要移动和/或复制被模型化的类型吗？

·哪种转换是可接受的？哪种是需要的？

·设想中的基本操作集是模板无关的吗？例如，能在操作时使用\*=或\*和=吗？

这里，同样的，archetypes（28.3节，655页）的概念很有用。

### E.4.2 概念粒度

随着concept成为C++语言的一部分，建立“concept库”的想法自然会出现，就像我们构建类库和模板库时那些特性就可用一样。正如其他库一样，将我们的concepts以多种方式分层也是很自然的。我简要的讨论一下迭代器类别的例子，并且展望我们可以在其一旁建立“范围类别”并在其顶部建立“序列concepts”等等也不是什么巨大的飞跃。

另一方面来说，我们可能被诱惑将所有这些concepts建立在“基础分析”concepts之上。例如，我们可以想象：

template<typename T, typename U>

concept Addable =

requires (T x, U y) {

x + y;

}

不推荐这样做，然而，因为这个concept没有清晰的语义内涵，分散类型会满足它。比如，当T和U都是std::string或一个类型是指针，另一个是整数类型，同样也可是算术类型时满足。在这三个例子中，Addable的概念意味着某些基本上都不同（各自来说，一系列关联的事物，迭代器错置，以及算术加法的变换）。因此介绍这个concept是一个具有模糊的接口和倾向于触发奇怪歧义的库的手册。

取而代之，很明显的是concepts是在问题领域出现的语义附注的最佳设计。以规则化的方式这样做与我们的库总体设计的改进绑定，此外他还会带来我们客户面前展现的接口的稳定性与清晰性。这和STL（Standard Template Library）加入到C++标准时十分相似。尽管它没有以语言为基础的“concept”来处理，很大部分的设计是在思想中的concept的观念（例如迭代器和迭代器分级），剩余的就是历史。

### E.4.3 二进制兼容性

有经验的C++编程者意识到当特定实体（特定的函数和成员函数）编译到低等级的机器码时，已声明的名称及其类型和实体的范围的组合与其有关。这个名称，通常与该实体的mangled name有关，被目标代码链接器用来解析实体的引用（例如，从其他对象文件）。例如，函数损坏名称的定义如下：

namespace X {

void f() {}

}

是\_ZN1X1fEv在因特尔Itanium C++ABI中[ItaniumABI]（在此编码中的X和f各自来自于名称空间名和函数名）。

Magled name无法在一个程序内“撞车”。因此，如果两个函数可以潜在的共存于一个程序，他们必须有不同的Magled name。也就意味着约束必须在函数名称内解码（因为除了约束和它们的函数体可以出现在不同的翻译单元中，模板特化在每种方式上都是独立的）。考虑一下两个翻译单元：

*#include* <iostream>

template<typename T>

concept HasPlus = requires (T x, T y) {

x+y;

};

template<typename T> int f(T p) requires HasPlus<T> {

std::cout << "TU1\n";

}

void g();

int main() {

f(1);

g();

}

和

*#include* <iostream>

template<typename T>

concept HasMult = requires (T x, T y) {

x\*y;

};

template<typename T> int f(T p) requires HasMult<T> {

std::cout << "TU2\n";

}

template int f(int);

void g() {

f(2);

}

程序必须输出：

TU1

TU2

也意味着两个f()的定义必须不同的mangled。