现代C++语言核心特性解析 - C++23标准补充

Version: 1.0 Release date: 2023.12.6

-1. 版权声明

c. 以本书为基础的衍生作品必须使用相同的许可协议。
2. 任何以商业目的使用本书的人必须在使用的同时获得**作者的书面许可**。

除非适用法律要求或书面同意，否则作者不提供任何形式的担保，明示或暗示，包括但不限于对适销性或特定用途的适用性的担保。作者对使用本书所导致的任何损失概不负责。

如有任何疑问，请联系moderncpp@163.com。

0. 前言

《现代C++语言核心特性解析》这本书是2021年10月出版的，到现在已经经过了2年多的时间，在此期间因为这本书我结识到一些志同道合的朋友，收到一些书友的认可和鼓励，并且在B站发布相关视频也引来了不少朋友的关注、点赞和收藏。可以说，这本书出版后带给我的惊喜，远远大过了出版这本书本身。

我很喜欢罗翔老师说的一段话，大概意思是：

> 人生95%的东西是我们决定不了的，如果真的取得了成就应该感谢自己以外的力量，因为自己能决定的事情太少了。这就是为什么如果真的获得了成就，应该积极回报这个社会，因为给你的不一定真的属于你。

我年初的时候就在想，用什么方式感谢支持我的书友们，想来想去也只能是将C++23标准的语言特性的解析作为礼物回馈给大家。当然，我也有其他的考虑，C++23标准的语言特性篇幅不多，不适合单独出版成书，也不应该加到原本的书中出第二版，它本身就更适合作为电子书分享出来。

如何读这本书呢？很简单，读自己感兴趣的部分即可。我很认同书友们对这本书的评价，它是一本工具书，需要的时候拿出来翻一翻就是一种很合适它的阅读方法。但是，有一点还是需要注意的，这本电子书作为《现代C++语言核心特性解析》的补充，它必然与本体有着比较大的联系，有的章节会提到本体的一部分内容，所以对于不熟悉C++11到C++20标准的朋友来说，读起来可能会有一些不太友好。

最后来到传统的感谢环节，首先肯定是感谢支持我的朋友们，正是因为你们的支持我才会有动力继续编写《现代C++语言核心特性解析》的后续内容。然后，是感谢爱人、父母对我的包容，让我在闲暇的时候可以专心写作而不需要关心生活琐事。最后的5%，感谢一下自己的坚持吧。

关于本书的后续更新都会在 https://github.com/0cch/moderncpp_public 上进行发布，书中的任何问题都可以通过以下三种方式联系到我：

1. 在此Repository中提交issue；
2. 在微信公众号【现代CPP随笔】上留言；
3. 从微信公众号获取微信群二维码，加入C++讨论群。

目录

1. 支持预处理指令 `#elifdef` 和 `#elifndef`

一直以来，C++为了保持与C语言的兼容性会快速跟进C语言新增特性。这次也不例外，C23标准通过了加入`#elifdef` 和 `#elifndef` 预处理指令的提案，于是C++23标准也加入了`#elifdef` 和 `#elifndef` 这两个预处理指令。

我们知道，在C++23标准之前就已经支持了许多条件预处理指令了，其中就包括：

```
#if expr
#ifdef identifier
#ifndef identifier
#elif expr
#else
#endif
```

比较有趣的是

```
#if expr
#elif expr
#else
#endif
```

已经可以很好的完成条件预编译的任务了，但是标准还引入了`#ifdef identifier`和`#ifndef identifier`，显然这里的目的是为了简化代码，这两个预处理指令完全可以被替换为：

```
#if defined(identifier)
#if !defined(identifier)
```

当然了，虽然说有代替的写法，但是对于我来说还是倾向少敲一些字符的。于是，跟我有相同想法的人就会提出疑惑，既然有`#ifdef identifier`和`#ifndef identifier`，为什么不提供对应的`#elif`呢？并且这部分的实现并不困难。

现在，C23和C++23标准满足了上述需求，提供了

```
#elifdef identifier
#elifndef identifier
```

两个预处理指令，我们可以将其看作

```
#elif defined(identifier)
#elif !defined(identifier)
```

的简化版本。

注意，两种预处理指令的混用并不会造成不良后果，比如：

```
#if FOO
#elifdef BAR
#else
#endif
```

是不会有任何问题的，就如同我们现在写以下代码：

```
#ifdef BAR
#elif FOO
#else
#endif
```

最后来看C23标准中一段典型的用法：

```
#ifdef __STDC__
#define TITLE “ISO C Compilation”
#elifndef __cplusplus
#define TITLE “Non-ISO C Compilation”
#else /* C++ */
#define TITLE “C++ Compilation”
#endif
```

2. 允许重复属性

C++23标准允许属性列表序列中存在重复属性。（注意：该特性为C++11缺陷报告，所以编译器实现可以将其加入到C++11标准。）例如：

```
[[noreturn, noreturn]] void foo() {throw;}

int main()
{
    foo();
}
```

使用GCC10和Clang12编译上面的代码会提示错误信息：

```
error: attribute 'noreturn' can appear at most once in an attribute-list
```

因为在属性列表中存在重复属性`noreturn`，所以较老版本的编译器会报错。由于该缺陷报告是2020年7月提出的，因此使用较新版本的编译器可以成功编译，即使使用C++11标准。

允许属性列表存在重复属性的理由很简单，因为在缺陷修复之前，虽然属性列表中不能声明重复属性，但是重复属性可以声明在不同的属性列表，例如：

```
[[noreturn]][[noreturn]] void foo() {throw;}

int main()
{
    foo();
}
```

上面这段代码在任何支持属性的编译器版本都可以编译成功，我们会发现`[[noreturn]][[noreturn]]`和`[[noreturn, noreturn]]`具有完全相同的含义，但前者可以通过编译而后者不行，这显然是不合理的，该缺陷的修改有助于属性列表的行为表现一致。

3. 允许`static_assert`声明在与求值无关的模板上下文

缺陷报告CWG2518对许`static_assert`声明做了一些改进，在CWG2518提出之前，下面这段代码一定会编译失败：

```
template <class T>
void f(T t)
{
    static_assert(false);
}
```

原因很简单，`static_assert(false);`静态断言失败，导致编译器拒绝编译。显然，这样是不合理的。不合理的原因是，这段代码只是简单的定义了函数模板`f`，并没有对其进行实例化，在这种情况下对其进行诊断是没有必要的。

为了解决上述问题，CWG2518对`static_assert`声明做了一些修改，标准规定：

> 在`static_assert`声明中，常量表达式在上下文中被转换为bool类型，转换后的表达式应为常量表达式。如果表达式转换后的值为`true`，或者表达式是在模板定义的上下文中求值的，则声明无效。否则，`static_assert`声明将失败，程序将无法编译，并产生诊断信息。

注意这段描述中提到：

> 或者表达式是在模板定义的上下文中求值的，则声明无效。

也就是说，如果只在模板定义中声明`static_assert`，那么声明是无效的。因此，我们如果使用GCC13和Clang17编译这段代码是可以顺利编译通过的。

让我们把这条规则延申一下，下面这段代码使用CWG2518改进后的标准也能够成功编译：

```
constexpr void use(int x){};

template <class T>
void f(T t)
{
    if constexpr (sizeof(T) == sizeof(int)) {
        use(t);
    }
    else {
        static_assert(false, "must be int-sized");
    }
}

void g(char c)
{
    f(0);
}
```

首先，我们知道定义函数模板`f`是不会触发静态断言失败的，然后在实例化过程中`if constexpr`的特点是只编译符合条件的分支，这里`f(0)`推导出`T`的类型是`int`，`sizeof(T) == sizeof(int)`的计算结果为`true`，`else`分支不会编译，也就是说`static_assert(false, "must be int-sized");`这句代码不会被编译和诊断，编译器顺利完成编译。当然，如果我们将函数模板调用从`f(0)`修改为`f(c)`，那么就会导致静态断言失败，从而引发编译报错。

4. assume 属性

Clang和MSVC都支持一种内部函数，该函数提供了一种方法优化程序代码的生成，简单来说程序员可以给定一个表达式并假设该表达式的运算结果为`true`，编译器会根据这个假设对代码进行优化，以便编译器生成更快、更小的代码，这个功能在高性能、低延迟计算中是非常有用的。

我们先看看Clang中如何使用这个内置函数优化代码：

```
int divide_by_32(int x) {
    __builtin_assume(x > 0);
    return x/32;
}
```

其中`__builtin_assume`是内置假设函数，该函数假设`x > 0`为`true`以帮助编译器优化代码。让我们来对比一下使用假设内置函数和没有使用该函数的汇编代码（编译使用O3优化选项）：

```
; 使用__builtin_assume(x > 0);
divide_by_32(int):
        mov     eax, edi
        shr     eax, 5
        ret
```

对比

```
; 没有使用__builtin_assume(x > 0);
divide_by_32(int):
        lea     eax, [rdi + 31]
        test    edi, edi
        cmovns  eax, edi
        sar     eax, 5
        ret
```

可以看出第一份代码因为我们假设`x > 0`，所以编译器没有为`x`为负数的情况生成代码，直接采用逻辑右移完成计算。相反，第二份代码需要区分`x`的正负符号，若`x`是正值，`cmovns  eax, edi`执行数据移动，并且做算术右移，相当于直接对`edi`寄存器的值算术右移5位；若`x`是负值，`cmovns  eax, edi`不执行数据移动，并且做算术右移，相当于对`rdi + 31`的值算术右移5位。

通过上述例子，相信读者朋友已经了解了内置假设函数功能的实用之处。但需要注意的是，使用这个功能必须对假设条件有足够清晰的判断，例如上面的例子中，我们必须确定`x`是大于0的，否则造成的后果是编译器优化了负数逻辑，导致当`x`是负数的时候函数计算出错。

可以说假设功能是个专家级功能，需要谨慎使用，若使用方法正确，可以取得非常不错的优化效果。不过可惜的是，该功能一直没有标准进行规范，都是编译器自定义实现的，例如：

- MSVC使用`__assume(expr );`
- Clang使用`__builtin_assume(expr);`
- GCC没有这个功能，但是可以用内置函数`__builtin_unreachable();`简单模拟：

```
if (expr ) {} else { __builtin_unreachable();}
```

我们可以使用一个宏来满足移植要求：

```
#if defined(__clang__)
    #define ASSUME(expr) __builtin_assume(expr)
#elif defined(__GNUC__)
    #define ASSUME(expr) if (expr) {} else { __builtin_unreachable(); }
#elif defined(_MSC_VER)
    #define ASSUME(expr) __assume(expr)
#endif
```

不过，这样的解决方案并不完美，比如GCC方案的代码中，`expr`会在运行时进行求值计算的，而Clang和MSVC则不会，但是Clang和MSVC的内置函数对于`expr`的要求也略有区别，所以很显然我们需要一种标准化的方案来完成内置假设函数的工作。

assume 属性语法

C++23标准提出了一个统一的方案解决上述的问题，即添加一个新的`assume`属性。使用`assume`作为属性名的理由非常简单直接，因为内置函数就是使用的`assume`，所以它对于主流编译器的用户来说不会陌生，很容易了解它是用来干什么的。

这里读者朋友可能会提出疑问，原本编译器使用的都是内置函数执行假设功能，为何标准选择属性来完成此功能呢？这是一个非常值得探讨的问题，让我们回想一下C++标准推出新功能的一般方法吧。

首先，假设C++引入一个新的关键字，假定就是`assume`关键字吧，语法为`assume(expr);`。看起来不错，但是这里的问题之一是引入新的关键字会影响老版本编译器或者不支持新关键字编译器的兼容性，另一个问题是引入新关键字对语言来说是一个非常重大的修改，而假设功能是一个专家级功能，使用它的时候必须非常小心谨慎，如果只是一小部分人使用的功能，那么新增一个关键字是不合适的。综合看来，引入关键字是不合适的。

其次，可以假定C++引入一个标准库函数，例如`std::assume(expr)`，语法上看是库函数调用，实际上`expr`并不求值。这样做最大的问题是，我们引入了一个看似库函数，但是行为模式和库函数完全不同的“新类型”函数。它的行为和内置函数一样，你不能获取它的函数地址，不能分配函数指针，与其说它是一个函数，不如说是一个带有命名空间的关键字。显然，这种假定也不太合适。

最后，宏方案就更不必说了，C++一直致力于减少宏的使用，没有任何理由让C++引入一个新的宏。

所以总体来说，`assume`作为一个新的属性引入进来是最合适的。因为，首先，属性并不影响老版本编译器和不支持该属性的编译器的兼容性，因为标准规定对于实现不支持的属性可以直接忽略该属性。这条规定正好又符合假设的另外一个特点，即忽略假设和忽略属性一样，都不会影响程序的正常逻辑。其次，属性一般来说影响的都是代码的生成和优化，例如`likely`、`unlikely`、`noreturn`等，这些是针对编译器的后端而不是前端，假设功能正好也是属于代码优化功能。总结来说，`assume`作为属性对语言的影响最小，并且最符合假设功能的特点，所以`assume`应该是一个属性。

好了，解释了`assume`被定义为属性的原因，现在让我们来看看它的具体语法：

```
[[assume(expr)]];
```

这里`assume`属性只能用于空语句，简单来说就是属性声明之后直接跟随`;`。

`expr`必须是一个条件表达式，该表达式可以根据上下文转换为`bool`，并且总是假设表达式的计算结果为`true`。如果表达式的计算结果确实为`true`，那么`assume`属性不会对代码逻辑产生任何影响（性能优化，逻辑不变），否则会产生未定义的行为。产生未定义行为的逻辑很好理解，我们假设了一个错误的条件，编译器根据错误条件的优化是无法预测的。

值得注意的是，因为`expr`是一个条件表达式，所以赋值表达式，逗号表达式都是错误语法：

```
void f() {}
bool g(bool x)
{
    [[assume(x = true)]];   // 编译失败
    [[assume(f(), x)]];     // 编译失败
    f();
    return x;
}
```

当然，要让以上代码编译成功也很简单，只需要使用一对括号包括表达式即可：

```
void f() {}
bool g(bool x)
{
    [[assume((x = true))]]; // 编译成功
    [[assume((f(), x))]];   // 编译成功
    f();
    return x;               // 直接编译为 return true;
}
```

这里必须强调一下`[[assume((f(), x))]];`的逻辑，因为它很容易让人理解为，假设`f()`和`x`都必须返回`true`。实际上并不是这样的，这里的假设是运行函数`f()`之后的副作用导致`x`返回值为`true`。

需要注意的是，`assume`属性中的条件表达式虽然不会求值，但是引用的变量和函数等必须在有其声明的上下文中，是存在潜在求值的。

```
int divide_by_32(int x) {
    [[assume(y > 0)]];  // 编译失败，不存在y的定义
    return x/32;
}
```

上面的代码会导致编译报错，GCC会明确提示：

```
'y' was not declared in this scope
```

规则很简单，但是还没有结束，我还想给大家展示一个细节，请看下面的代码：

```
constexpr auto f(int i) {
    return sizeof([=] {});
}
static_assert(f(0) == 1);   // lambda表达式类型的结构大小为1
```

对比使用假设属性后：

```
constexpr auto f(int i) {
    return sizeof([=] { [[assume(i==0)]]; });
}
static_assert(f(0) == 4);   // lambda表达式类型的结构大小为4
```

神奇的情况发生了，我们刚才强调过，`assume`属性并不会对表达式求值，但是lambda表达式的大小却被改变了。这是为什么呢？其实，上文中已经提到过了：“引用的变量和函数等必须在有其声明的上下文中”。请注意，这两份代码中，前者因为的lambda表达式在函数体没有捕获任何变量，所以相当于`[]{}`这个lambda表达式，于是lambda表达式的字节大小为1。而后者则不同，为了能够编译成功，lambda表达式必须捕获变量`i`，相当于`[i]{}`，所以它的字节大小为4。

这个问题影响大么？事实上并不大，首先作为属性影响数据结构内存布局和大小的`assume`并不是第一个，`no_unique_address`也是其中之一，可能区别是`no_unique_address`的作用就是影响数据结构的内存布局和大小。其次，`assume`没有影响到代码的语义，程序在语义层面没有变化。当然，也不是完全没有影响，至少如果lambda表达式作为ABI，这个改变是会被观察到的。

最后，需要说明一下关于常量表达式函数中使用假设的情况，如果在常量表达式函数内部，我们遇到了一个假设，但在假设的条件表达式无法进行常量求值，例如：

```
int foo() {
    return 0;
}
constexpr int bar()
{
    [[assume(foo() == 0)]];
    return 1;
}
int main() {
    constexpr int retval = bar();
    return retval;
}
```

结论是，上述代码应该能够顺利的编译通过。进行常量求值时，若无法在编译时检查假设，则应忽略该假设，而不是让编译器报错。

无法推导假设的条件表达式的情况

我们刚刚举到的例子都是比较简单的条件表达式，例如：

```
int divide_by_32(int x) {
    [[assume(x > 0)]];
    return x/32;
}
```

这里直接给出了参数`x`是大于0的假设，优化没有任何问题。如果将表达式复杂化一点呢？

```
int divide_by_32(int x) {
    [[assume(-x < 0)]];
    return x/32;
}
```

使用GCC13编译这份代码也不会有问题，可以正确的优化。因为虽然条件表达式不能求值，但是标准允许分析表达式的形式并推断出用于优化程序的信息。对于上述这种确定性的表达式，编译器可以根据实际情况推导优化信息。

当然，现实世界的编译器无法在所有情况下执行所需的转换，这并没有关系，因为标准规定如果编译器无法获取任何有用的优化信息，那么可以忽略该假设。例如：

```
#include <iostream>
int divide_by_32(int x) {
    [[assume((std::cin >> x, x > 0))]];
    return x/32;
}
```

上面的代码中，很明显`x`的值是无法在编译期确定的，也就无法推断出有用的信息，所以GCC在这里的做法是忽略该假设，不对代码做任何优化。

请注意，这里还存在一个争议点，那就是既然`x > 0`是假设的一部分，那么是否应该假设`std::cin >> x`这个输入值就是大于0的呢？这个论点是有道理的，例如我明确知道外部输入是由另外一个程序执行，并且输入的数字一定大于0，那么这个假设就是有意义的，编译器应该对它进行优化，将出现未定义行为的责任抛给程序员。

事实上，标准并没有对这种情况做明确规定，编译器完全可以根据实际情况做出选择，使用完全不同的策略，其中就包括忽略假设。

5. 初始化语句允许别名声明

我们知道，在现代C++代码中，一直都提倡使用别名声明`using`来代替`typedef`，但是C++23标准之前有一处却有例外：

```
#include <vector>

int main()
{
    std::vector<int> vec{ 0, 1, 2 };
    for (typedef int T; T i : vec) {}
}
```

上面这份代码使用C++20标准可以编译成功，但是如果修改为：

```
#include <vector>

int main()
{
    std::vector<int> vec{ 0, 1, 2 };
    for (using T = int; T i : vec) {}
}
```

新版本的Clang和GCC虽然可以编译成功但是会给出警告，而旧版本编译器会直接报错。使用C++23标准则没有上述问题，C++23标准扩展了初始化语句以允许别名声明，这其中包括`if`、`switch`和基于范围的`for`循环语句中的声明。

6. 允许在lambda表达式上使用属性

我们常说，lambda表达式是一种匿名函数对象，它可以在被调用的位置或作为参数传递给函数的位置定义，比起函数对象要方便很多。但是一直以来，函数对象有一个功能是lambda表达式不具备的，那就是我们可以给函数对象的函数调用运算符函数附加属性，例如：

```
struct Functor {
    [[nodiscard]] int operator() () { return 42; }
};

int main()
{
    Functor f;
    auto retval = f(); // 不能忽略返回值
}
```

在上面的代码中，我们可以自定义的附加`Functor`的函数调用运算符函数的属性，例如这里我们附加了`nodiscard`属性，这样一来如果我们在调用`f()`的时候忽略其返回值，那么编译器会给出警告提示我们需要获取函数对象的返回值。

现在问题来了，如果是编写lambda表达式函数，我们怎么能够给它附加属性呢？能够这样写么？

```
int main()
{
    auto lm = [[nodiscard]][]{ return 42; }; // 编译报错
    auto retval = lm();
}
```

显然上面的代码是无法编译通过的，所以结论是在C++23标准以前我们无法给lambda表达式的函数调用运算符函数附加属性。

幸运的是，上述问题在C++23标准中得到了解决，在C++23标准中可以通过新的语法来给lambda表达式的函数调用运算符和运算符模板附加属性了，其语法也非常简单，在lambda表达式引导符即捕获列表之后可以紧跟一个属性声明，例如：

```
int main()
{
    auto lm = [][[nodiscard]]{ return 42; };
    auto retval = lm(); // 不能忽略返回值
}
```

在上面的代码中，空捕获列表`[]`之后紧跟着一个`[[nodiscard]]`，这样lambda表达式的函数调用运算符函数就被声明了`nodiscard`属性，如果调用`lm()`的时候没有接收返回值，那么编译器会给出对应的警告信息。

值得一提的是C++标准委员会在添加这个特性的时候，对于属性声明应该添加在lambda表达式的什么位置也是经过非常深入的思考的。例如，像`Functor`的函数调用运算符函数那样将属性声明在lambda表达式的开头显然不可取，这样会造成语法的解析问题。那么最合适的位置既然不能是开头，但是又想和普通函数保持一致，那么只能放在函数名和参数列表之间，例如：

```
struct Functor {
    int operator() [[nodiscard]] () { return 42; }
};

int main()
{
    auto retval = Functor()();
}
```

在上面的代码中，`Functor`的函数调用运算符函数的属性`nodiscard`也是生效的。所以对于lambda表达式而言，将属性声明放在捕获列表和参数列表之间也是最合适的地方了。

```
auto lm = [][[noreturn]](){ throw; };  // 属性声明在捕获列表[]和参数列表()之间
```

7. 引入auto(x)和auto{x}代替decay-copy

C++23标准为`auto`占位符引入了一个新的功能，即使用 `auto(x)` 和 `auto{x}` 将 `x` 转换为纯右值。该特性最重要的任务之一就是实现decay-copy功能，所谓decay-copy就是将一个值转换为其退化的纯右值副本，可以简单实现为：

```
template <typename T>
constexpr std::decay_t<T> decay_copy(T&& v) noexcept(
    std::is_nothrow_convertible_v<T, std::decay_t<T>>)
{
    return std::forward<T>(v);
}
```

关于decay-copy的意义，我们先来看一个例子：

```
class thread {
   public:
    template <class Function, class Arg>
    thread(Function&& f, Arg&& arg) {
        // ...
        std::invoke(std::forward<Function>(f),
                    std::forward<Arg>(arg));
        // ...
    }
};
```

上面这段代码表达的是创建一个线程并且执行目标函数的过程，其中用到了完美转发保证参数属性的一致性。但是这里有一个参数生命周期的问题，由于传递参数使用了完美转发，函数内无法控制外部引用的生命周期，也就是说在线程执行的过程中这些参数可能会无效导致未定义行为。要解决这个问题，就可以用到decay-copy：

```
class thread {
   public:
    template <class Function, class Arg>
    thread(Function&& f, Arg&& arg) {
        // ...
        std::invoke(decay_copy(std::forward<Function>(f)),
                    decay_copy(std::forward<Arg>(arg)));
        // ...
    }
};
```

通过decay-copy的方法，创建了参数的纯右值副本，这样生命周期就在新线程的执行函数中进行控制，不会出现变量失效导致未定义行为的问题。实际上，STL中的`thread`类也使用了这种技巧，但没有明确定义decay-copy的函数：

```
class thread {
// ...
    template <class _Fn, class... _Args>
    void _Start(_Fn&& _Fx, _Args&&... _Ax) {
        using _Tuple                = tuple<decay_t<_Fn>, decay_t<_Args>...>;
        auto _Decay_copied          = _STD make_unique<_Tuple>(_STD forward<_Fn>
(_Fx), _STD forward<_Args>(_Ax)...);
        constexpr auto _Invoker_proc = _Get_invoke<_Tuple>(make_index_sequence<1
+ sizeof...(_Args)>{});

        _Thr._Hnd =
            reinterpret_cast<void*>(_CSTD _beginthreadex(nullptr, 0,
_Invoker_proc, _Decay_copied.get(), 0, &_Thr._Id));

        if (_Thr._Hnd) {
            (void) _Decay_copied.release();
        } else {
            _Thr._Id = 0;
            _Throw_Cpp_error(_RESOURCE_UNAVAILABLE_TRY_AGAIN);
        }
    }
// ...
}
```

上面的代码引用自MSVC的STL，可以看到虽然代码中没有明确调用一个`decay_copy`函数，但是实际上在声明`_Decay_copied`时完成了相同的操作。显然，在这份代码中decay-copy的代码是比较繁琐的，如果在语言层面具备这种功能那么就回简单得多，这也是使用`auto(x)`和`auto{x}`代替decay-copy的意义所在。

另一个需要在语言层面代替`decay_copy`函数的原因在于，使用函数总是会有其局限性，例如提案文档中的例子：

```
class A {
    int x;

public:
    A();

    auto run() {
        f(A(*this));           // 编译成功
        f(auto(*this));        // 编译成功
        f(decay_copy(*this));  // 编译失败
    }

protected:
    A(const A&);
};
```

在上面的代码中，因为类`A`的构造函数被声明为`protected`，那么`decay_copy`也没办法在这种情况下使用，毕竟我们不能让`decay_copy`成为每个类的`friend`，但是如果使用语言层面的`auto`就不会存在这种限制。

还有一个`decay_copy`从语义层面不能做到的是，调用此函数必然会进行拷贝操作，生产纯右值副本，这里我们不考虑编译器的优化。但是`auto`则不同，它针对已经是纯右值的实体可以不做任何操作从而提高效率。

关于`auto(x)`和`auto{x}`创建纯右值副本的另一个有趣的问题是，`decltype(auto(expr))`和`std::decay_t`有什么区别？来看看下面两份代码：

```
auto p = std::make_unique<char>();
static_assert(std::is_same_v<std::decay_t<decltype(p)>, std::unique_ptr<char>>);
```

以及

```
auto p = std::make_unique<char>();
static_assert(std::is_same_v<decltype(auto(p)), std::unique_ptr<char>>);
```

它们在编译时会有区别么？答案是前者只是简单的做类型变换，比如将`T&`变换为`T`，并没有被类型本身的定义约束，而后者则不同，`auto`会根据上下文环境对类型进行判断，比如在上面的代码中，因为`std::unique_ptr<char>`是没有拷贝构造函数的，所以这里会编译失败，GCC提示信息为：

```
error: use of deleted function 'std::unique_ptr<_Tp, _Dp>::unique_ptr(const
std::unique_ptr<_Tp, _Dp>&) [with _Tp = char; _Dp = std::default_delete<char>]'
```# 更改lambda表达式后置返回类型的解析范围

返回类型后置是lambda表达式的基本语法，这个语法在C++23标准之前存在一个问题，即返回类型解析范围依赖于外部，比如下面这个例子：

``` C++
auto counter = [j=0]() mutable -> decltype(j) {
    return j++;
};
```

上面这段代码中，`counter`是一个使用了初始化捕获（init-capture）的lambda表达式，它的返回值类型为`decltype(j)`，读者可以思考一下，这份代码是否可以编译成功？答案是不能，因为外部并不存在`j`的声明，而lambda表达式内部的`j`对于`decltype(j)`来说是不可见的。在绝大多数情况下，这不会造成什么问题，毕竟编译报错会给程序员提示并修复自己的程序，但是也有例外情况：

```
double j = 4.2;
auto counter = [j=0]() mutable -> decltype(j) {
    return j++;
};
```

在声明了外部的`double j = 4.2;`后，lambda表达式`counter`可以编译成功了，但是却产生了一个问题。这里`decltype(j)`的`j`使用了外部的变量声明，也就是说`counter`的返回类型为`double`，而lambda表达式内部基于初始化捕获的`j`是一个`int`类型。我们可以使用以下代码来验证这个结论：

```
#include <type_traits>
int main()
{
    double j = 4.2;
    static_assert(std::is_same_v<decltype(j), double>);
```

```
    auto counter = [j=0]() mutable -> decltype(j) {
        static_assert(std::is_same_v<decltype(j), int>);
        return j++;
    };

    auto i = counter();
    static_assert(std::is_same_v<decltype(i), double>);
}
```

以上代码使用MSVC 19.35、Clang 16和GCC 13.1均可编译成功。

上述问题不仅存在于使用初始化捕获的lambda表达式，其他lambda表达式同样存在这样的问题，只是触发它并不容易：

(下面内容需要等待Clang17后续版本验证，Clang17.0.1目前还有问题)

```
template <typename T> int bar(int&, T&&);
template <typename T> void bar(int const&, T&&);

int main()
{
    int i = 0;
    auto f = [=](auto&& x) -> decltype(bar(i, x)) {
        return bar(i, x);
    };
    f(42); // 编译失败
}
```

上面这段代码调用`f(42)`会编译失败，因为返回类型推导中`decltype(bar(i, x))`中的`i`是lambda表达式外部的`int i = 0;`的`i`，所以其类型为`int`，调用`int bar(int&, T&&)`函数，返回`int`类型。而`return bar(i, x);`语句中的`i`是lambda表达式捕获的，由于lambda表达式没有声明为`mutable`，因此这里的`i`的类型为`const int i`，于是调用`void bar(int const&, T&&)`函数，返回类型为`void`。lambda表达式`f`声明的返回类型和实际返回类型不一致，导致编译失败。

从C++23标准开始，上述问题得以解决，其解决方案简单来说就是后置返回类型中的任何可捕获实体的行为应该像被捕获一样，无论它最终是否被捕获。

```
int i;
auto f = [=](int& j) -> decltype((i)) {
    return j;
};
```

例如上面的代码，在C++23标准之前，`f`的返回类型为`int&`。从C++23标准开始，`f`的返回类型为`const int&`，即使lambda表达式实际上并没有捕获它。

值得注意的是，这种特殊的待遇只适用于后置返回类型中的可捕获实体，若可捕获实体出现在函数参数范围之前或位于参数声明子句中，则会导致编译错误。

为了更明确新规则带来的影响，我们来看看下面这份代码：

```
void f()
{
    float x = 0.;
    [=]<decltype(x) P> {};   // 编译错误：x可被捕获，但在lambda的函数参数之前
    [=](decltype((x)) y) {}; // 编译错误：x可被捕获，但在lambda的函数参数声明子句中
    [=]
    {
        []<decltype(x) P> {};         // 编译成功：x没有被捕获
        [](decltype((x)) y) {};       // 编译成功：x没有被捕获
        [x = 1](decltype((x)) z) {}; // 编译错误：x是可被捕获，但在lambda的函数参数参数
声明子句中
    };
}
```

代码的注释很明确的解释了lambda表达式在C++23标准中编译成功或者失败的原因。总结来说，如果一个可捕获的实体，比如例子中的局部变量`x`，声明在lambda的函数参数之前或者声明在参数声明子句中，就会导致编译失败。但是如果该实体是不可捕获的，例如捕获列表为空`[]`，那么代码就可以编译成功。

8. `char8_t`兼容性和可移植性修复

根据上一节的内容，我们了解了自C++20标准引入UTF-8字符类型以后，出现了与C++17标准不兼容的情况，例如：

```
const char* ptr0 = u8"hello utf-8";             // C++17：编译成功 | C++20：编译失败
const unsigned char* ptr1 = u8"hello utf-8";    // C++17：编译失败 | C++20：编译失败
const char arr0[] = u8"hello utf-8";            // C++17：编译成功 | C++20：编译失败
const unsigned char arr1[] = u8"hello utf-8";   // C++17：编译成功 | C++20：编译失败
constexpr const char* resource_id () {
    return u8"◠‿◠";                              // C++17：编译成功 | C++20：编译失
败
}
```

这个兼容问题的影响面比想象的要大，导致不少代码库在升级到C++20的时候无法成功编译。为了解决这个问题，编译器实现给出了编译选项消除影响，例如，Clang和GCC可以通过`-fno-char8_t`或者MSVC通过`/zc:char8_t-`告诉编译器在使用C++20标准时，让`char8_t`特性退回到C++17标准，以保证源代码可以顺利编译。

除了通过编译器实现来解决上述问题，聪明的C++程序员也想到了模板元编程的办法，例如：

```
#include <utility>

template<std::size_t N>
struct char8_t_string_literal {
    static constexpr inline std::size_t size = N;

    template<std::size_t... I>
    constexpr char8_t_string_literal(const char8_t (&r)[N],
std::index_sequence<I...>)
    : s{r[I]...} {}

    constexpr char8_t_string_literal(
    const char8_t (&r)[N])
```

```
    : char8_t_string_literal(r, std::make_index_sequence<N>()) {}

    auto operator <=>(const char8_t_string_literal&) const = default;

    char8_t s[N];
};

template<char8_t_string_literal L, std::size_t... I>
constexpr inline const char as_char_buffer[sizeof...(I)] =
    { static_cast<char>(L.s[I])... };

template<char8_t_string_literal L, std::size_t... I>
constexpr auto& make_as_char_buffer(std::index_sequence<I...>) {
    return as_char_buffer<L, I...>;
}

constexpr char operator ""_as_char(char8_t c) {
    return c;
}

template<char8_t_string_literal L>
constexpr auto& operator""_as_char() {
    return make_as_char_buffer<L>(std::make_index_sequence<decltype(L)::size>());
}

#if defined(__cpp_char8_t)
#   define U8(x) u8##x##_as_char
#else
#   define U8(x) u8##x
#endif

int main () {
    constexpr const char* p = U8("text");
    constexpr const char r = U8('a');
    return 0;
}
```

已经熟悉模板元编程和语言新特性的朋友理解上面这段代码应该没有问题，但是对于新手朋友可能会比较困难，我这里简单解释一下。

1. `char8_t_string_literal` 是一个字面量类模板，其模板参数 `N` 由构造函数 `constexpr char8_t_string_literal(const char8_t (&r)[N])` 推导而来，这里使用的是类模板的模板实参推导特性。
2. `char8_t_string_literal(const char8_t (&r)[N])` 没有直接进行构造，而是调用 `template<std::size_t... I> constexpr char8_t_string_literal(const char8_t (&r)[N], std::index_sequence<I...>)` 完成对象的构造，这里使用的是委托构造函数的特性。
3. 代理构造函数通过可变模板参数的包展开特性初始化了数据成员 `char8_t s[N];`，完成对象的构造。
4. `as_char_buffer` 是通过变量模板的特性将 `char8_t` 数组转换为 `char` 数组。
5. `make_as_char_buffer` 使用常量表达式函数和可变模板参数包展开的特性来构造 `as_char_buffer`。
6. `operator""_as_char` 使用用户自定义字面量的特性针对字符和字符串调用不同的函数。

7. 最后通过`__cpp_char8_t`功能测试宏特性，在不同的编译环境下预处理为不同的代码。

如果读者朋友现在看不懂这份代码也没关系，上述提的特性在本书中都会详细说明，读完本书后再回头理解这份代码应该就不会有问题。

这份代码对兼容的实现方式虽然很"神奇"，但是正如我之前所说的，它不利于新手理解，属于C++专家热衷的代码写法。并且使用宏来区分编译环境也并不是一个好主意，这很容易让不熟悉编码环境的程序员落入代码陷阱。就像Windows编程中`TCHAR`和`TEXT(...)`，程序员在编写C++代码时必须了解这些宏和编译环境的关系以保证字符串被正确处理。顺便提一句，Windows编程应该尽量使用Unicode版本的API，它们通常有`W`后缀，这样就不会受到代码页改变的影响了。

引入`char8_t`类型后，C++20标准除了和C++17标准产生了兼容性问题，与C语言的兼容性也受到了影响：

```
extern const char* a = u8"hello utf-8";      // C：编译成功 | C++20：编译失败
extern const char b[] = u8"hello utf-8";     // C：编译成功 | C++20：编译失败
extern const unsigned char* c = u8"hello utf-8";     // C：编译成功 | C++20：编译失败
extern const unsigned char d[] = u8"hello utf-8";    // C：编译成功 | C++20：编译失败
```

为了缓解这类兼容性问题，C++23标准对`char8_t`的转换规则做出了一些修改（该修改也作为C++20的缺陷报告提出，因此新版本编译器的C++20标准也支持该修改），标准规定：`char`或`unsigned char`数组可以通过UTF-8字符串字面量或用大括号括起来的字符串字面量进行初始化。即下面的例子中，部分代码可以重新编译成功了：

```
const char* ptr0 = u8"hello utf-8";              // 编译失败：指针类型无法转换
const unsigned char* ptr1 = u8"hello utf-8";     // 编译失败：指针类型无法转换
const char arr0[] = u8"hello utf-8";             // 编译成功：数组转换
const unsigned char arr1[]{u8"hello utf-8"};     // 编译成功：数组转换
```

注意，标准只是规定`char`或`unsigned char`数组可以被UTF-8的字符串字面量初始化，而没有提到对应的指针类型的转换，所以指针类型的转换依旧是非法的。不过，这个修改已经足以解决大部分问题，使用指针的代码只需要稍作修改就能兼容新老版本的编译环境。至于指针之间的转换，因为涉及到包括`u8""`如何退化为`const char*`或`const unsigned char*`，重载的排序是什么，以及在什么情况下适用等大量的工作，所以截至C++23标准为止，并未对其做出修改。

最后需要注意的是，因为允许UTF-8的字符串字面量隐式转换为`char`或`unsigned char`数组，所以对于包含数组的结构体初始化的重载解析会带来一个问题：

```
struct A {
    char8_t s[10];
};
struct B {
    char s[10];
};

void f(A);
void f(B);

int main() {
    f({u8""});
}
```

上面这份代码，可以使用Clang16和GCC12编译成功，因为适配缺陷报告之前编译器认为`{u8""}`只能用于初始化结构体`A`，所以重载决议很明确不会有问题。但是标准修改之后，结构体`A`和`B`都可以使用`{u8""}`进行聚合初始化，这就造成了调用函数`f`时的二义性问题。好在，这个问题的触发条件比较罕见，我们也不必为此过于担心了。

9. 引入翻译字符集

C++23标准引入了翻译字符集，简单来说翻译字符集就是翻译时使用的抽象字符集，它是可以表示所有有效通用字符名称的等效字符。

翻译字符集具体范围包括：

- 具有 ISO/IEC 10646 命名，并且有唯一的 UCS(Universal Coded Character Set) 标量值标识的每个字符，以及
- 未分配命名的每个 UCS 标量值的独立字符。

翻译字符集的作用在于，过去编译器翻译的第一阶段，是由编译器的实现来定义源文件的字符映射到基础字符集的方式，这里的问题在于基础字符集是一个非常小的集合，那么如何映射不在基础字符集的字符就成了一个问题。从C++23标准开始，新的规定是源文件的字符应该映射到翻译字符集，这个字符集范围要大得多，基础字符集是它的一个子集，这样就解决了上述问题。

举例来说，在C++23之前，一个编译器可以实现成这样：

```
#define S(x) # x
const char * s1 = S(Köppe);       // "K\\u00f6ppe"
const char * s2 = S(K\u00f6ppe);  // "K\\u00f6ppe"
```

因为`ö`这个字符不是基础字符集，所以可能发生转义`\u00f6`。现在根据C++23标准，`ö`是可以被识别的字符，那么结果就应该是：

```
#define S(x) # x
const char * s1 = S(Köppe);       // "Köppe"
const char * s2 = S(K\u00f6ppe);  // "Köppe"
```

然后，我们必须提到一个有趣的事实，现实情况是几乎所有的主流编译器实现都已经实现了后者的功能，并没有提供转义UCN。# 一致的字符字面量编码

在C++23标准之前，预处理器条件中的字符字面量的行与C++表达式中的行为是不一定一致的，例如：

```
#if 'A' == '\x41'
//...
#endif
```

与

```
if ('A' == 0x41) {}
```

判断的结果可能会有所不同。因为C++23之前的标准中明确表示，两种代码的行为是否一致是由实现定义的。也就是说编译器可以决定上述代码的行为结果。

不过，对于我们程序员来说，当然是更希望它们有一致的行为结果，因为这种特性可以用于检测字符编码，例如sqlite头文件sqliteInt.h中的这段代码：

```
/*
** Check to see if this machine uses EBCDIC.  (Yes, believe it or
** not, there are still machines out there that use EBCDIC.)
*/
#if 'A' == '\301'
# define SQLITE_EBCDIC 1
#else
# define SQLITE_ASCII 1
#endif
```

所以，C++23标准采纳了这种实践和用户期望，规定预处理器条件中的字符字面量的行为应该与C++表达式中的行为一致。

10. consteval if 语句

前面的章节介绍了，C++20标准引入了`consteval`说明符和`std::is_constant_evaluated()`函数让我们能够方便的编写常量求值的代码。简单回顾一下这两个特性，`consteval`说明符用于只能在常量求值期间调用的函数，也叫做立即函数；而`is_constant_evaluate()`则是一个库函数，用于检查当前求值计算是否为常量求值。不过遗憾的是，这两个功能却有些不太兼容，请注意以下代码：

```
consteval int f(int i) { return i; }

constexpr int g(int i) {
    if (std::is_constant_evaluated()) {
        return f(i) + 1; // <==
    } else {
        return 42;
    }
}

consteval int h(int i) {
    return f(i) + 1;
}
```

这是提案文档中了一份代码，使用C++20标准编译这份代码会出现编译错误。原因很简单，函数`g`中`f(i) + 1`不是一个常量表达式，其中`i`是不确定的。这里我们会发现，`std::is_constant_evaluated()`完全没起到想要的作用，代码的本意是如果`if`的条件确定是常量求值，那么编译器才应该编译`if`中的这段代码，而不是直接报错。顺带一提，函数`h`是立即函数，所以调用函数`f`不会有任何问题。

很显然，上面代码的`std::is_constant_evaluated()`没有到达我们预期的效果，为了解决这个问题，C++23标准引入了新的 consteval if 语句，它的语法比较独特：

```
if consteval {} else {}
```

注意这里`if`后面的`consteval`是不带括号的，而且随后的大括号也是强制不可忽略的，同样`else`之后的大括号也是不可忽略的，也就是说它们必须都是复合语句。当然了，`else`整体可以省略。语法表达的含义是：如果代码执行的上下文在常量求值期间发生，则执行`if`子语句，否则，如果存在选择语句的`else`部分，则执行`else`子语句。

有了新的consteval if语句后，我们就可以修改上面的代码为：

```
consteval int f(int i) { return i; }

constexpr int g(int i) {
    if consteval {
        return f(i) + 1; // ok: immediate function context
    } else {
        return 42;
    }
}

consteval int h(int i) {
    return f(i) + 1; // ok: immediate function context
}
```

使用C++23标准可以顺利的编译这段代码，不会出现之前的编译错误。值得一提的是，我们可以通过consteval if语句来编写一个自己的`is_constant_evaluated`函数：

```
constexpr bool is_constant_evaluated() {
    if consteval {
        return true;
    } else {
        return false;
    }
}
```

当然我并不推荐这么做，统一使用标准库才是更好的选择，提案文档的作者也表示不应该弃用`std::is_constant_evaluated()`函数，即使从现在看来正确的实现该函数并不困难，但是每个人都去实现一份相同的代码显然不是一个好的选择，另外使用标准提供的`std::is_constant_evaluated()`函数能够让编译器更准确的提供警告信息，例如：

```
constexpr int g() {
  if constexpr (std::is_constant_evaluated()) { // warning: always true
    return 42;
  }
  return 7;
}
```

编译以上代码会输出如下警告信息：

```
warning: 'std::is_constant_evaluated' will always evaluate to 'true' in a
manifestly constant-evaluated expression
```

再次强调一下，consteval if语句中的大括号是不能忽略的，以下代码编译器会提示缺少括号`{`：

```
constexpr int g(int i) {
    if consteval {
        return f(i) + 1;
    } else
        return 42;
}
```

编译以上代码，编译器会提示：

```
error: expected { after else
```

至于为何这样强制规定，提案文档上也给了解释：

> There is no technical reason to mandate braces. Our reason for them is to emphasize visually that we're dropping into an entirely different context.

很明确，并没有技术上的理由，就是为了在视觉上强调代码正在进入一个完全不同的环境而已，而这样做的很重要的一部分原因是，`if consteval`中`consteval`没有用括号包括起来，括号具有很明显的分割视觉的作用。如果允许忽略大括号，那么就可能出现以下这种代码：

```
if consteval for (; it != end; ++it) ;
if consteval if (it != end);
```

显然，这样的代码看起来会很头痛。那么问题又来了，为何不将语法规定为`if (consteval)`呢？关于这一点，提案作者也给了简单明了的回答：

> we think the fact that if consteval looks different from a regular if statement is arguably a benefit.

总之，作者认为这种与众不同的语法是有好处的，而且C++委员会也接受了这样的提案。

最后来介绍一下consteval if语句的否定形式：

```
if not consteval { }

// 或者

if ! consteval { }
```

注意，标准中强调

```
if ! consteval compound-statement
```

本身不是一个consteval if语句，但是它等价于consteval if语句：

```
if consteval { } else compound-statement
```

同理

```
if ! consteval compound-statement1 else statement2
```

本身也不是一个consteval if语句，但是它等价于consteval if语句：

```
if consteval statement2 else compound-statement1
```

11. 分隔的转义序列

转义序列是C++语言的基本功能，我们常用的转义序列包括简单转义序列`\r`、`\n`、`\t`等，数字转义序列八进制的`\nnn`（1-3个八进制数）、十六进制`\xn...`（任意个十六进制数）以及通用字符转义序列`\unnnn`（小写u和4个十六进制数）、`\Unnnnnnnn`（大写U和8个十六进制数）。

其实对我来说，C++的转义序列已经足够简单实用了，但是C++委员会的专家还是提出了两个问题。

首先，通用字符转移序列规定大写字母U和小写字母u后必须紧跟8个和4个十六进制数，但是我们使用的Unicode代码空间为 0-0x10FFFF，大部分情况下我们用不到8个十六进制数，例如`\U0001F1F8`，就需要填充3个无意义的0，是有改进空间的。

其次，八进制和十六进制的数字转义序列由于有可变的序列长度，所以使用起来容易出现一些不易察觉的错误。例如，`\17`会被解析为一个数值`0x0f`，但是如果写成`\18`那么它会被解析为数值`0x01`和字符`'8'`。为了解决这个问题，微软的提出了一些解决方案，比如使用宏，例如：

```
#define Bell '\x07'
```

通过宏来避免转义序列和字符的混淆。

另外，微软还提出可以使用断开字符串的技巧来区分转义序列和字符，例如：

```
"\xabc"     // 一个字符
"\xab" "c"  // 两个字符
```

这两个方法对解决上述问题确实有效，但是C++委员会认为这样还不够优雅，所以提出了新的分隔方案。

C++23标准规定：

- 使用`\o{}`的语法来定义一个八进制转义序列，其中`{}`中可以有任意个八进制数；
- 使用`\x{}`的语法来定义一个十六进制转义序列，其中`{}`中可以有任意个十六进制数；
- 使用`\u{}`的语法来定义一个通用字符转义序列，其中`{}`中可以有任意个十六进制数，当然`{}`中的数值必须是一个有效的Unicode变量值。

所以上述例子中`"\xab" "c"`可以表示为`"\x{ab}c"`。注意，分隔转义序列的处理发生在字符串连接之前，所以我们必须保证分隔转义序列的完整性，例如：

```
char str[] = "\x{4" "2}";
```

上面的代码是会导致编译报错的，编译器会提示`'\x{'`不是由`'}'`结束。

```
error: '\x{' not terminated with '}' after \x{4
```

值得一提的是，分隔的转义序列的这个语法和另一个新语言特性是密切相关的，那就是具名通用字符转义。

12. 显式对象参数

自从C++11标准发布以来，C++的新标准一直致力于减少代码中的冗余，例如`auto`占位符、基于范围的for循环等等。但是由于右值引用的出现，有一部分代码却比过去显得更加冗余了，那就是成员函数的重载。

对象的不同类型引发的成员函数重载问题

一般来说，成员函数可以有CV限定符，因此可能存在特定类同时需要特定成员函数的`const`和非`const`重载的情况。当然，也可能需要`volatile`重载，但这不太常见，为了描述简洁我们可以忽略它。在有const重载的情况下，两个函数基本上做的是相同的任务，唯一的区别是函数访问的对象是否为常量类型。我们可以通过复制函数的实现代码来处理，例如，STL中`vector`的运算符函数`operator`

`[]`：

```
// https://github.com/microsoft/STL/blob/main/stl/inc/vector

_Ty &operator[](const size_type _Pos) noexcept
{
    auto &_My_data = _Mypair._Myval2;
    return _My_data._Myfirst[_Pos];
}

const _Ty &operator[](const size_type _Pos) const noexcept
{
    auto &_My_data = _Mypair._Myval2;
    return _My_data._Myfirst[_Pos];
}
```

或者将一个重载委托给另一个，比如可以将上述代码改写为：

```
_Ty &operator[](const size_type _Pos) noexcept
{
    return const_cast<_Ty &>(
        static_cast<const vector &>(*this)[_Pos]
    );
}

const _Ty &operator[](const size_type _Pos) const noexcept
{
    auto &_My_data = _Mypair._Myval2;
    return _My_data._Myfirst[_Pos];
}
```

在上面这段代码中，非`const`版本的`operator[]`运算符函数委托调用了`const`版本的`operator[]`运算符函数。虽然这份代码没有上一段直观，但是更容易维护。

不过话说回来，无论是复制代码还是委托调用，似乎都不是很好的解决方案，尤其是C++11引入了右值引用以后。本来只需要维护2个重载函数，现在变成了4个：`&`、`const&`、`&&`和`const&&`。

我们有三种方案来编写这4个函数，包括上文中提到的两种，分别是：

1. 实现4个重载函数，这样不可避免的会出现重复代码；
2. 将其中3个重载函数委托给另外1个重载函数，这种方法可以避免大量重复的代码，但是过多的类型转换让代码看起来并不优雅；
3. 实现一个辅助函数模板，将4个重载函数委托到该辅助函数模板，让编译器来推导调用对象的真实类型，避免编写类型转换的代码。相对于前两种方法，这种方法更加简洁明了。

让我们来看看提案文档中提供的一个例子，该例子使用了上述3种方法实现了`optional<T>::value()`的重载集：

```
//
// 实现4次重载函数
//
template <typename T>
class optional {
  // ...
```

```
  constexpr T& value() & {
    if (has_value()) {
      return this->m_value;
    }
    throw bad_optional_access();
  }

  constexpr T const& value() const& {
    if (has_value()) {
      return this->m_value;
    }
    throw bad_optional_access();
  }

  constexpr T&& value() && {
    if (has_value()) {
      return move(this->m_value);
    }
    throw bad_optional_access();
  }

  constexpr T const&&
  value() const&& {
    if (has_value()) {
      return move(this->m_value);
    }
    throw bad_optional_access();
  }
  // ...
};

//
// 3个重载函数委托给另外1个重载函数
//
template <typename T>
class optional {
  // ...
  constexpr T& value() & {
    return const_cast<T&>(
      static_cast<optional const&>(
        *this).value());
  }

  constexpr T const& value() const& {
    if (has_value()) {
      return this->m_value;
    }
    throw bad_optional_access();
  }

  constexpr T&& value() && {
    return const_cast<T&&>(
      static_cast<optional const&>(
        *this).value());
  }
```

```
  constexpr T const&&
  value() const&& {
    return static_cast<T const&&>(
      value());
  }
  // ...
};

//
// 实现辅助函数模板
//
template <typename T>
class optional {
  // ...
  constexpr T& value() & {
    return value_impl(*this);
  }

  constexpr T const& value() const& {
    return value_impl(*this);
  }

  constexpr T&& value() && {
    return value_impl(move(*this));
  }

  constexpr T const&&
  value() const&& {
    return value_impl(move(*this));
  }

private:
  template <typename Opt>
  static decltype(auto)
  value_impl(Opt&& opt) {
    if (!opt.has_value()) {
      throw bad_optional_access();
    }
    return forward<Opt>(opt).m_value;
  }
  // ...
};
```

可以看到，第三种方法已经比较接近理想的情况了，但是依然摆脱不了必须重复实现4个重载函数的窘境。当然了，聪明的读者可能已经想到了解决方法——使用一个友元非成员函数模板即可：

```
template <typename T>
class optional {
    // ...
    template <typename Opt>
    friend decltype(auto) value(Opt&& o) {
        if (o.has_value()) {
            return forward<Opt>(o).m_value;
        }
        throw bad_optional_access();
    }
    // ...
};
```

这样的解决方案当然是可行的，不仅如此，它被使用起来也不会觉得繁琐。但是不要忘了，我们最初是打算用成员函数的方法解决成员函数的重载问题，那么现在是时候介绍C++23标准引入的新特性显式对象参数了。

显式对象参数语法

事实上，显式对象参数这个特性在提案之初并不叫这个名字，在提案的时候它叫做可推导`this`。其实在我个人看来，后者理解起来更加容易。所谓可推导`this`就是指`this`指针对象的类型是可以被推导出来的。

语法形式为：声明一个非静态成员函数或者函数模板，其第一个参数是一个显式对象参数，用前缀关键字`this`表示，具体形式为：

```
struct X {
    void foo(this X const& self, int i);

    template <typename Self>
    void bar(this Self&& self);
};
```

可以看到上面这段代码的第一个成员函数`foo`，它的第一个参数类型是`X const&`，而在类型声明之前有关键字`this`，所以参数`self`就是显式对象参数。需要注意的是，单纯使用显式对象参数的非静态成员函数并没有什么意义，无法在重载时让编译器帮助我们推导对象。我们需要的是使用了显式对象参数的非静态成员函数模板，例如函数模板`bar`。可以看到函数模板`bar`的参数类型为模板参数`self&&`，在`self&&`之前有关键字`this`，指示该函数为显式对象参数的函数模板。阅读过右值引用章节的朋友肯定马上反应过来了，这里使用了万能引用的特性，代码通过这个特性让编译器去推导对象的具体类型。可以看出，显式对象参数的语法只是结合了关键字`this`、模板和万能引用，没有任何额外新知识的引入，所以理解起来还是很容易的。

在实现显式对象参数的非静态成员函数时必须使用显式对象参数来引用成员，例如：

```
struct B {
    int i = 0;
    template <typename Self> int foo1(this Self&& self) { return i; }
    template <typename Self> int foo2(this Self&& self) { return this->i; }
    template <typename Self> int foo3(this Self&& self) { return self.i; }
};

B b;
b.foo1();   // 编译成功：无法引用非静态成员B::i
b.foo2();   // 编译失败：无法引用this指针
b.foo3();   // 编译成功
```

观察上面的代码可以发现，无论是直接访问数据成员`i`，还是通过`this`指针访问`i`都是不可行的，编译器会直接报错。在这种情况下，我们只能通过`self`来访问数据成员`i`。

值得注意的是，具有显式对象参数的成员函数不能是`static`或`virtual`，此外它们也不能具有cv或者引用限定符。

```
struct X {
    virtual void foo(this X const& self, int i);
    // C7678: 具有显式对象参数的成员函数不能是虚函数

    static void foo1(this X const& self, int i);
    // C7669: 具有显式对象参数的函数不能声明为“静态”

    void foo2(this X const& self, int i) const;
    void foo3(this X const& self, int i) &&;
    // C7672: 具有显式对象参数的成员函数无法跟随对象参数说明符
};
```

违反上述语法规定，MSVC会给出非常明确的错误提示。

接下来，让我们看看显式对象参数是如何使用的：

```
struct D : X { };

void ex(X& x, D const& d) {
    x.foo(42);       // 参数self绑定到X，i的数值为42
    x.bar();         // 根据引用折叠原理，Self推导类型为X&，所以调用为X::bar<X&>
    move(x).bar();   // 根据引用折叠原理，Self推导类型为X，所以调用为X::bar<X>

    d.foo(17);       // 参数self绑定到d，类型依然为X const&，i的数值为17
    d.bar();         // 根据引用折叠原理，Self推导类型为D const&，所以调用为X::bar<D
const&>
}
```

从这段代码可以看出，具有显式对象参数的非静态成员函数的调用方法和普通非静态成员函数相同，也就是说显式对象作为函数的第1个实参是由编译器自动输入，而不需要我们在编写代码时输入，而手动输入的第1个实参会作为函数的第二个参数，依此类推。

还有一点需要注意，这里函数模板`bar`虽然看上去是`X`的成员，但`Self`的最终推导类型为`D const&`，也就是推导出了`X`的派生类型`D`。

现在我们可以用这个新特性改写`optional<T>::value()`了：

```
template <typename T>
struct optional {
  template <typename Self>
  constexpr auto&& value(this Self&& self) {
    if (!self.has_value()) {
      throw bad_optional_access();
    }

    return forward<Self>(self).m_value;
  }
};
```

这段代码用1个具有显式对象参数的函数模板实现了4个重载函数，无论是编写代码还是维护代码都更加简单明了了。

由于在使用上具有显式对象参数的函数和普通成员函数是相同的，这也引申出另外一个问题，如果通过两种方式声明了相同的函数会发生什么情况呢？答案是编译出错。

```
struct X {
    void bar()&&;
    void bar(this X&&); // 编译失败：无法重载
};
```

上述代码在编译的时候报错，因为具有显式对象参数的成员函数`void X::bar(this X &&)`无法重载具有隐式对象参数的成员函数`void X::bar(void) &&`。请注意，这里两个函数必须具有相同的参数和相同的对象参数限定符，如果稍有不同则不会出现编译错误：

```
struct Y {
    void bar() &;
    void bar() const&;
    void bar(this Y&&); // 编译成功
};
```

调用具有显式对象参数的函数指针

现在，我们已经知道了具有显式对象参数的非静态成员函数的调用方法和普通非静态成员函数相同，那么使用函数指针调用具有显式对象参数的函数是否也和普通非静态成员函数相同呢？答案是不同。

```
struct Y {
    int foo(int, int) const&;
    int bar(this Y const&, int, int);
};

Y y;
y.foo(1, 2);    // 编译成功
y.bar(3, 4);    // 编译成功

auto pf = &Y::foo;
pf(y, 1, 2);              // 编译失败
(y.*pf)(1, 2);            // 编译成功
std::invoke(pf, y, 1, 2); // 编译成功

auto pg = &Y::bar;
```

```
pg(y, 3, 4);               // 编译成功
(y.*pg)(3, 4);             // 编译失败
std::invoke(pg, y, 3, 4); // 编译成功
```

观察上面的代码就会发现，具有显式对象参数的函数指针更加类似于静态成员函数指针，需要显式传递对象参数`pg(y, 3, 4);`，这一点与非静态成员函数指针有着显著的区别需要读者注意一下。

静态函数还是非静态函数

相信读者阅读到这里已经产生了一个疑惑，具有显式对象参数的成员函数到底是一个静态成员函数还是非静态成员函数呢？将其定义为静态成员函数显然不行，它的函数调用方式和非静态成员函数一样，但是如果将其定义为非静态成员函数，它的函数指针调用又更加类似静态函数，更进一步来说显式对象参数的成员函数不能直接访问`this`指针，也没办法隐式引用`this`指针，所以其行为也更像静态函数。

显然这是一个比较具有争议的话题，无论如何定义都不会有一个完美的结果，能做的只是将其影响缩减到最小。根据提案文档的建议，具有显式对象参数的成员函数应该被定义为非静态成员函数，我们可以认为普通的非静态成员函数是一个具有隐式对象参数的成员函数。

原因主要考虑到两个方面：

1. 如果将具有显式对象参数的成员函数定义为静态函数，那么是否需要引入`static`关键字，这样将引入更多问题。
2. 具有显式对象参数的成员函数采取的是非静态函数的调用方式，需要在对象上调用，这是绝大多数的情况。并且对于调用者来说，并不关心其是函数是显式参数对象还是隐式参数对象。

所以基于这些考虑，提案给出了明确的函数属性：

1. 具有显式对象参数的成员函数是非静态成员函数；
2. 具有显式对象参数的成员函数不能声明为`static`；
3. 具有显式对象参数的成员函数不能是`virtual`；
4. 传统的非静态成员函数将被称为隐式对象成员函数。

具有显式对象参数的lambda表达式

显式对象参数还有一个有趣的地方，那就是可以在lambda表达式上使用：

```
#include <iostream>

int main()
{
    auto fib1 = [](this auto&& self, int n) -> decltype(auto) {
        if (n < 2) {
            return n;
        }
        return self(n - 1) + self(n - 2);
    };

    auto fib2 = [] <class Self>(this Self && self, int n) -> decltype(auto) {
        if (n < 2) {
            return n;
        }
        return self(n - 1) + self(n - 2);
    };
```

```
    std::cout << "fib1()=" << fib1(6) << " fib1()=" << fib1(6) << std::endl;
}
```

上面这段代码使用了泛型lambda表达式，这个特性在lambda表达式章节有过介绍。除此之外，可以看到在lambda表达式的参数中使用了显式对象参数，有了这个参数我们可以完成lambda表达式的递归操作，调用方法如代码所示`self()`。

有的读者这里可能会提出疑问，在实际编码过程中使用到递归的场景本就不多，而在lambda表达式中使用递归的情况更是少之又少，那么具有显式对象参数的lambda表达式的实际意义是否就不大了呢？当然不是，让我们来考虑另外一个场景：

```
#include <string>
struct Functor {
    std::string str;
    template<typename Self>
    auto&& operator() (this Self&& self)
    {
        return std::forward<Self>(self).str;
    }
};

int main()
{
    Functor f{"hello"};
    auto y1 = f();
    auto y2 = std::move(f)();
}
```

在上面的代码中，`Functor`是一个典型的函数对象，当然也是一个具有显式对象参数的函数对象。我们在调用函数对象的适合，编译器会根据函数对象的不同类型调用不同的构造函数，`auto y1 = f();`因为`f`是左值，所以调用的是`std::string`的拷贝构造函数，而`auto y2 = std::move(f)();`中因为`std::move(f)`是一个右值，所以调用了`std::string`的移动构造函数。

而对于lambda表达式来说，本质上实现的功能就是函数对象，如果不支持显式对象参数就无法完美的代替函数对象了，所以让lambda表达式支持显式对象参数是非常有必要的。

```
#include <string>

int main()
{
    std::string str{"hello"};
    auto f = [str]<typename Self>(this Self&& self) {
        return std::forward_like<Self>(str);
    };
    auto y1 = f();
    auto y2 = std::move(f)();
}
```

上面的这段代码使用具有显式对象参数的lambda和使用`Functor`完成了相同的功能，唯一需要注意的是，在lambda表达式中使用的是函数模板`std::forward_like`而非`std::forward`。`std::forward_like`是C++23标准库新引入的函数模板，功能简单来说就是将参数`str`的类型属性替换为`Self`的类型属性，从而让捕获对象具有lambda表达式对象的类型属性。

传值的显式对象参数

到目前为止，我们看到了不少显式对象参数的例子，它们无一例外都是引用类型，不过标准并没有规定显式对象参数必须引用类型，也就是说传值也是可以的，例如：

```
struct less_than {
    template <typename T, typename U>
    bool operator()(this less_than, T const& lhs, U const& rhs) {
        return lhs < rhs;
    }
};

int main() {
    less_than{}(4, 5);
}
```

这个功能在小类型的参数传递上可以优化性能，例如`std::string_view`，因为它足够小到可以通过CPU寄存器来传递数据，就像`int`和`double`类型，并且编译器能够排除外部干扰，从而更好的优化代码。实际上，任何不需要修改对象本身的小类型都可以使用这种技巧来优化性能，包括上述代码中的`less_than`，因为使用传值的方法就避免了隐式引用临时对象。

为了让读者更好的理解这种优化，让我们深入到汇编看看以下这段代码：

```
// /O1 /Ob0

struct implicit_ref {
    int x;
    int y;
    int operator() (int z)
    {
        return x + y + z;
    }
};

struct explicit_ref {
    int x;
    int y;
    int operator() (this explicit_ref self, int z)
    {
        return self.x + self.y + z;
    }
};

int main() {
    implicit_ref{3, 4}(5);
    explicit_ref{3, 4}(5);
}
```

为了让`implicit_ref`和`explicit_ref`的`operator()`函数得到充分优化的同时又不会发生内联，这里需要使用MSVC编译参数`/O1 /Ob0`。我们会获得以下汇编代码：

```
int implicit_ref::operator()(int) PROC
        mov     eax, DWORD PTR [rcx+4]
```

```
        add     eax, DWORD PTR [rcx]
        add     eax, edx
        ret     0
int implicit_ref::operator()(int) ENDP

static int explicit_ref::operator()(this explicit_ref,int) PROC
        mov     rax, rcx
        shr     rax, 32
        add     eax, ecx
        add     eax, edx
        ret     0
static int explicit_ref::operator()(this explicit_ref,int) ENDP

main    PROC
        xor     eax, eax
        ret     0
main    ENDP
```

从这段汇编代码可以看出，`implicit_ref::operator()(int)`的汇编代码需要访问栈上的内存，而`explicit_ref::operator()(this explicit_ref,int)`则只需要访问CPU的寄存器，效率高低在对比之下是一目了然的。

私有继承问题

使用具有显式对象参数的函数还存在一个私有继承的问题，这个问题和模板推导是有关系的，请看下面的代码：

```
class Base {
    int i;
public:
    template <typename Self>
    auto&& get1(this Self&& self) {
        return std::forward<Self>(self).Base::i;
    }

    int get2() { return i; }
};

class Derived : private Base {
    double i;
public:
    using Base::get1;
    using Base::get2;
};

int main () {
    Derived().get1();   // 编译失败，推导为Derived试图访问Base私有变量i
    Derived().get2();   // 编译成功
}
```

在这段代码中，虽然派生类`Derived`是私有继承的`Base`，但是它通过`using Base::get2;`将函数引入到了派生类，所以这里`Derived().get2();`可以编译成功。而`Derived().get1();`就没有那么幸运了，由于模板推导的关系，`Self`推断为`Derived`而非`Base`，`Derived`访问`Base`的私有变量导致编译失败。

现在知道了问题的原因，那么我们应该如何解决该问题呢？其实很简单，做一次类型转换即可。

```
class Base {
    int i;
public:
    template <typename Self>
    auto&& get1(this Self&& self) {
        return std::forward_like<Self>((Base&)self).i; // 类型转换
    }

    int get2() { return i; }
};
```

类型转换的代码很简短，但是却有两个地方需要注意。

1. 需要用到C风格的类型转换，因为C风格的类型转换会无视访问检查。如果采用C++风格的类型转换，这里会编译失败。
2. 需要使用`std::forward_like<Self>`进行完美转发，让`self`的类型属性和`Self`保持一致，否则`self`会被编译器当作左值引用处理。

优化CRTP

写过几年C++的朋友多多少少都接触过这样一个特殊的设计模式，它将派生类型作为模板参数传递给基类模板作为实现静态多态性的一种方式，这个设计模式就是奇异递归模板模式（Curiously Recurring Template Pattern，简称CRTP）。这个模式最初给我留下深刻印象的应用场景是ATL（Active Template Library），还记得当时我被这种模板的使用方法震撼了好久。震撼的原因就是因为这种使用方法过于“奇异”，对于初学者来说理解起来并不容易。例如下面这段代码：

```
template <typename Derived>
struct add_postfix_increment {
    Derived operator++(int) {
        auto& self = static_cast<Derived&>(*this);
        Derived tmp(self);
        ++self;
        return tmp;
    }
};

struct some_type : add_postfix_increment<some_type> {
    int n = 0;

    some_type& operator++() {
        ++n;
        return *this;
    }
    using add_postfix_increment::operator++;
};

int main()
{
    some_type t;
    t++;
    ++t;
```

```
}
```

这段代码实现了一个添加后缀递增运算符函数的类模板`add_postfix_increment`，该模板接受一个派生类模板参数，并且调用派生类的前缀递增运算符函数实现后缀递增运算符函数。不得不说，这个实现方法确实挺绕的。如果想优化这段代码，我们就需要用到显式对象参数了。

```
struct add_postfix_increment {
    template<typename Self>
    auto operator++(this Self&& self, int) {
        auto tmp = self;
        ++self;
        return tmp;
    }
};

struct some_type : add_postfix_increment {
    int n = 0;

    some_type& operator++() {
        ++n;
        return *this;
    }
    using add_postfix_increment::operator++;
};

int main()
{
    some_type t;
    t++;
    ++t;
}
```

可以看到，新的`add_postfix_increment`不再是一个类模板，所以在定义`some_type`的时候不需要将自己的类型传递给基类，编译器会根据调用对象推导出`Self`的具体类型。虽然整体代码并没有减少很多，但是更容易让人理解了。

如果这份代码还无法让读者体会到使用显式对象参数的优势，那么我们可以再看一个提案文档中更加复杂的例子，这个例子使用建造者模式实现了一个构建器：

```
#include <iostream>
#include <string>

struct Builder {
    std::string str;
    Builder& a() { str += "a()"; return *this; }
    Builder& b() { str += "b()"; return *this; }
    Builder& c() { str += "c()"; return *this; }
};

int main()
{
    std::cout << Builder().a().b().a().b().c().str;
}
```

构建器使用了一段时间后出现了新的需求，需要引入新操作`d()`和`e()`。这是我们会发现，简单的在`Builder`上修改是有问题的，可能会引入兼容性方面的隐患。从`Builder`上派生一个新的构建器类可能是不错的选择，不过这也有一个问题，就是`a()`、`b()`和`c()`三个函数返回的是基类对象，无法调用`d()`和`e()`，所以还这里需要用到CRTP来进行类型转换。

```
#include <iostream>
#include <string>

template <typename D = void>
struct Builder {
    using Derived = std::conditional_t<std::is_void_v<D>, Builder, D>;
    Derived& self() {
        return *static_cast<Derived*>(this);
    }

    std::string str;
    Derived& a() { str += "a()"; return self(); }
    Derived& b() { str += "b()"; return self(); }
    Derived& c() { str += "c()"; return self(); }
};

struct Special : Builder<Special> {
    Special& d() { str += "d()"; return *this; }
    Special& e() { str += "e()"; return *this; }
};

int main()
{
    std::cout << Builder().a().b().a().b().c().str;
    std::cout << Special().a().d().e().a().str;
}
```

可以看到，一旦我们开始对CRTP做更多的事情，代码的复杂度就会迅速增加。这里大部分复杂度来自于使用模板来选择类型的操作，即`std::conditional_t<std::is_void_v<D>, Builder, D>;`，当模板参数`D`是`void`时`Derived`的类型为`Builder`，反之`Derived`的类型为模板参数`D`。通过CRTP的方法获取到派生类的类型后，就可以通过类型转换`*static_cast<Derived*>(this);`，让基类的函数返回派生类的对象了。

如果将上面这段代码修改为基于显式对象参数的方法，那么就能忽略使用模板选择类型的操作，从而让代码更容易理解。

```
#include <iostream>
#include <string>

struct Builder {
    std::string str;

    template <typename Self>
    Self& a(this Self&& self) { self.str += "a()"; return self; }

    template <typename Self>
    Self& b(this Self&& self) { self.str += "b()"; return self; }

    template <typename Self>
```

```
    Self& c(this Self&& self) { self.str += "c()"; return self; }
};

struct Special : Builder {
    Special& d() { str += "d()"; return *this; }
    Special& e() { str += "e()"; return *this; }
};

int main()
{
    std::cout << Builder().a().b().a().b().c().str;
    std::cout << Special().a().d().e().a().str;
}
```

显然，这份代码容易理解多了。不过这还没完，可能是提案作者觉得上述代码还不够有说服力，所以他对构建器又增加了需求，新增一个`f()`操作，结果代码就变成了：

```
#include <iostream>
#include <string>

template <typename D = void>
struct Builder {
    using Derived = std::conditional_t<std::is_void_v<D>, Builder, D>;
    Derived& self() {
        return *static_cast<Derived*>(this);
    }

    std::string str;
    Derived& a() { str += "a()"; return self(); }
    Derived& b() { str += "b()"; return self(); }
    Derived& c() { str += "c()"; return self(); }
};

template <typename D = void>
struct Special : Builder<std::conditional_t<std::is_void_v<D>, Special<D>, D>>
{
    using Derived = typename Special::Builder::Derived;
    Derived& d() { this->str += "d()"; return this->self(); }
    Derived& e() { this->str += "e()"; return this->self(); }
};

struct Super : Special<Super>
{
    Super& f() { str += "f()"; return *this; }
};

int main()
{
    std::cout << Builder().a().b().a().b().c().str;
    std::cout << Special().a().d().e().a().str;
    std::cout << Super().a().d().f().e().str;
}
```

好了，这份代码对于一些不熟悉模板元编程的读者来说已经有一些困难了，虽然原理上是一样的，但是有更多使用模板来选择类型的操作。为了让代码更容易理解，我们还是将其改写为基于显式对象参数的形式：

```
#include <iostream>
#include <string>

struct Builder {
    std::string str;

    template <typename Self>
    Self& a(this Self&& self) { self.str += "a()"; return self; }

    template <typename Self>
    Self& b(this Self&& self) { self.str += "b()"; return self; }

    template <typename Self>
    Self& c(this Self&& self) { self.str += "c()"; return self; }
};

struct Special : Builder {
    template <typename Self>
    Self& d(this Self&& self) { self.str += "d()"; return self; }

    template <typename Self>
    Self& e(this Self&& self) { self.str += "e()"; return self; }
};

struct Super : Special {
    template <typename Self>
    Self& f(this Self&& self) { self.str += "f()"; return self; }
};

int main()
{
    std::cout << Builder().a().b().a().b().c().str;
    std::cout << Special().a().d().e().a().str;
    std::cout << Super().a().d().f().e().str;
}
```

代码一目了然，编译器会自动推导`Self`为派生类型，不需要任何额外的模板元编程代码。因此我也推荐读者在试图编写CRTP的代码时，在编译环境允许的情况下，考虑使用显式对象参数，因为它是一个更好的解决方案。

对SFINAE友好

一直以来，重载对于SFINAE都是不友好的，在类型匹配的过程中，往往会出现我们意想不到的情况。这里以调用包装器`std::not_fn`为例，我们实现了其中函数调用运算符的两个重载：

```
template<class F>
struct not_fn_t
{
    F f;
    //
```

```
    // ...
    //
    template<class... Args>
    constexpr auto operator()(Args&&... args) &&
        noexcept(noexcept(!std::invoke(std::move(f), std::forward<Args>
(args)...)))
        -> decltype(!std::invoke(std::move(f), std::forward<Args>(args)...))
    {
        return !std::invoke(std::move(f), std::forward<Args>(args)...);
    }

    template<class... Args>
    constexpr auto operator()(Args&&... args) const&&
        noexcept(noexcept(!std::invoke(std::move(f), std::forward<Args>
(args)...)))
        -> decltype(!std::invoke(std::move(f), std::forward<Args>(args)...))
    {
        return !std::invoke(std::move(f), std::forward<Args>(args)...);
    }
};

template<class F>
constexpr not_fn_t<std::decay_t<F>> not_fn(F&& f)
{
    return { std::forward<F>(f) };
}
```

接着，我们编写两个不同的函数对象类型以及函数调用运算符函数模板的重载：

```
struct unfriendly {
    template <typename T>
    auto operator()(T v) {
        static_assert(std::is_same_v<T, int>);
        return v;
    }

    template <typename T>
    auto operator()(T v) const {
        static_assert(std::is_same_v<T, double>);
        return v;
    }
};

struct fun {
    template <typename... Args>
    bool operator()(Args&&...) = delete;

    template <typename... Args>
    bool operator()(Args&&...) const { return true; }
};
```

这两个函数对象类型各有特点，`unfriendly`的函数调用运算符函数中加入了`static_assert`，这意味着模板匹配失败会引发编译期断言。对于`fun`，如果试图实例化`operator()(Args&&...)`则会因为函数声明为`delete`编译失败。

现在我们结合上述两个函数对象类型和`not_fn`看看会发生什么。

```
int main()
{
    not_fn(unfriendly{})(1);    // 触发static_assert
    not_fn(fun{})();            // 编译成功
}
```

意想不到的事情发生了，`not_fn(unfriendly{})(1);`会触发`static_assert`，原因是实例化过程中，重载函数`auto operator()(T v) const`也会进行实例化，导致`std::is_same_v<T, double>`断言失败。无独有偶，`not_fn(fun{})();`在我们预期之外的编译成功了，我们的预期是因为`fun{}`，是非常量临时对象，所以应该实例化`bool operator()(Args&&...)`，但是编译器依然将重载函数模板都进行了实例化，导致编译器回退使用`bool operator()(Args&&...) const`让代码编译成功。

过去我们很难优雅的解决上述问题，但是现在可以使用显式对象参数来完成任务。

```
template <class F> struct not_fn_t {
    F f;

    template <class Self, class... Args>
    constexpr auto operator()(this Self &&self, Args &&...args)
noexcept(noexcept(
        !std::invoke(std::forward<Self>(self).f, std::forward<Args>(args)...)))
        -> decltype(!std::invoke(std::forward<Self>(self).f,
                                 std::forward<Args>(args)...))
    {
        return !std::invoke(std::forward<Self>(self).f,
                            std::forward<Args>(args)...);
    }
};

template<class F>
constexpr not_fn_t<std::decay_t<F>> not_fn(F&& f)
{
    return { std::forward<F>(f) };
}

//
// unfriendly和fun定义 ...
//

int main()
{
    not_fn(unfriendly{})(1);    // 编译成功
    not_fn(fun{})();            // 编译报错：找不到匹配的调用运算符
}
```

这里将类模板`not_fn_t`的调用运算符函数模板修改为具有显式对象参数的函数模板，并且再次编译代码。我们会发现这次`not_fn(unfriendly{})(1);`可以编译成功，而`not_fn(fun{})();`编译失败，MSVC虽然没有提示`bool operator()(Args&&...)`已经被删除，但是会提示找不到匹配的调用运算符，结果是符合我们预期的。之所以会有这样的结果，是因为在这次实例化的时候，减少了重载的干扰。对于`unfriendly`来说`not_fn_t`的调用运算符函数只有一个，编译器进行一次推导即可，不需要实

例化其他重载。而对于`fun`也一样，现在`not_fn_t`的调用运算符函数只有一个，`Self`被明确推导为`not_fn_t<fun>`（非`const`版本），这让代码必须确保`fun`的非`const`调用运算符是可用的。

总结

显式对象参数是一个有趣的章节，它的语法并不复杂，甚至可以说是非常简单，但是其应用确实比较复杂，理解起来会有一定难度。主要原因是这些应用都包括了对模板的使用，比如使用显示对象参数简化重载，替换CRTP，优化SFINAE等等，都涉及到模板的类型推导。不过即使理解起来不太容易，但熟悉以后的好处显而易见。此外，显式对象参数也有相对简单的应用，比如引用lambda表示自身，传值以优化性能等。所以综合来说，掌握这一章节是非常有必要的，尤其对于库作者更加重要，因为无论是重载、CRTP还是SFINAE都是现代C++代码库常用的技术。

13. 标识符语法使用UAX31

从C++23标准开始，标识符的语法开始使用Unicode 标准附件 31（Unicode Standard Annex 31）。这将导致一个有趣的影响：emoji字符不可用。在C++23标准之前，有一部分emoji是可用的，具体来说就是Unicode编码大于FFFF的字符，所以采用不同版本的编译器编译带有emoji的代码会有所区别，例如：

```
int ⏰ = 0; // <U+23F0> not valid
int 🕐 = 0; // <U+1F550>

int ☠ = 0; // <U+2620> not valid
int 💀 = 0; // <U+1F480>

int ✋ = 0; // <U+270B> not valid
int 👊 = 0; // <U+1F44A>

int ✈ = 0; // <U+2708> not valid
int 🚀 = 0; // <U+1F680>

int ☹ = 0; // <U+2639> not valid
int 😀 = 0; // <U+1F600>
```

使用Clang13编译上面的代码会发现小于FFFF的emoji字符会导致编译器报错：

```
error: non-ASCII characters are not allowed outside of literals and identifiers
int ⏰ = 0; //not valid
    ^~
error: expected unqualified-id
int ⏰ = 0; //not valid

...
```

而使用Clang14编译这份代码会发现所有的emoji字符都会导致编译器报错：

```
error: unexpected character <U+23F0>
int ⏰ = 0; //not valid
    ^~
error: expected unqualified-id
int ⏰ = 0; //not valid
      ^
error: unexpected character <U+1F550>
```

```
int 🕐 = 0;
    ^~
error: expected unqualified-id
int 🕐 = 0;

...
```

虽然在新标准中，标识符无法使用emoji，但是我认为这些改变只会影响类似国际混淆C代码竞赛（IOCCC）的代码。对于正常的项目应该说是毫无影响的，毕竟在字符串中声明这些emoji字符是不受影响的，例如：

```
#include <iostream>

int main()
{
    const char * str = "⏰🕐☠💀";
    std::cout << str << std::endl;
}
```

以上代码可以顺利的通过编译，并且输出正确的结果。

14. 允许复合语句末尾的标签(与C语言兼容)

2020年3月，C语言提案N2508扩展了标签在C语言能力，即C语言标签可以出现在复合语句的任何位置，包括声明之前和复合语句的末尾。为了保持与C语言的兼容性，C++在C++23标准也做了一个修改，即允许标签出现在复合语句末尾的位置，需要注意的是此前C++一直都支持将标签放在声明之前。

```
void foo(void)
{
first:      // C++支持，C23支持
    int x;
second:     // C++和C均支持
    x = 1;
last:       // C++23标准之前不支持，C23支持
}
```

解释一下这段代码，标签`first`的位置，也就是声明之前，在C++中一直都是支持的，而C语言需要用到C23标准。标签`second`的位置在C++和C语言中都是支持的。而`last`标签是C++23标准和C23标准开始支持的。

15. signed size_t 和 size_t 的字面量后缀 z 和 uz

在C++23标准之前，关于signed size_t 和 size_t一直都存在着一个不严重，但却让程序员感到不舒服的问题，这里先举个例子：

```
void setValue(int v);
void setValue(size_t v);

setValue(42);
```

这里的问题是，对于没有任何后缀的整型字面量，编译器会认为它的默认类型是`int`，如果想调用`size_t`版本的`setValue`函数，我们必须使用类型转换。

下面的代码应该也是经常会看到的：

```
std::vector<int> v{0, 1, 2, 3};
for (auto i = 0; i < v.size(); ++i) {
    std::cout << i << ": " << v[i] << '\n';
}
```

当需要用到数组下标的时候，往往容易编写以上代码。它的问题是，`auto`推导出的`i`的类型为int，而`v.size()`的返回类型是`size_t`，这就导致两种问题：

1. 由于`size_t`类型的定义是无符号的，所以这里存在有符号整数和无符号整数的对比问题，可能产生整型溢出。

```
开启选项 -Wall

<source>: In function 'int main()':
<source>:11:24: warning: comparison of integer expressions of different
signedness: 'int' and 'std::vector<int>::size_type' {aka 'long unsigned int'} [-
Wsign-compare]
   11 |     for (auto i = 0; i < v.size(); ++i) {
      |                      ~~^~~~~~~~~~
```

2. 由于`size_t`类型的长度与平台相关，所以这里还可能存在对比类型长度不一致的问题。

```
#ifdef _WIN64
    typedef unsigned __int64 size_t;
#else
    typedef unsigned int     size_t;
#endif
```

以上代码是MSVC上对于`size_t`的定义，可以看到虽然都是无符号，但是32位和64位代码`size_t`的定义是不同的。

当然，如果编写以下代码，编译器会直接报错：

```
for (auto i = 0, s = v.size(); i < s; ++i) {}
```

因为`i`被推导为`int`类型，导致`s`类型和`v.size()`的返回类型不一致。

除了在使用函数重载和`auto`容易出现上述问题以外，在使用模板的时候也容易出现这种问题。

```
std::max(42, v.size());
```

为了解决以上问题，C++23标准引入了z和uz两个后缀，用于直截了当的书写`signed size_t`和`size_t`类型的字面量。

```
std::vector<int> v{0, 1, 2, 3};
std::max(42uz, v.size());

for (auto i = 0zu, s = v.size(); i < s; ++i) {}

setValue(42uz);
```

上面的代码可以通过`uz`直接定义`size_t`类型的整型字面量，非常的简洁。值得注意的是，这里使用`uz`或者`zu`都是可以的，它们的含义相同。

相对于`uz`，`z`的作用会少很多，因为我们很少会使用`signed size_t`类型。当然也并不排除编写下面代码的可能：

```
std::vector<int> v{0, 1, 2, 3};
std::max(42z, std::ssize(v));
```

16. 可选的lambda表达式中的括号

在前面的章节介绍过，如果lambda表达式没有形参列表，那么括号是可选的，比如前面已经看到过的例子：

```
#include <iostream>

int x = 1;
int main()
{
    int y = 2;
    static int z = 3;
    auto foo = [y] { return x + y + z; };
    std::cout << foo() << std::endl;
}
```

可以看到，因为没有参数列表，这里忽略了括号也是可以的。但是，在C++23标准之前，忽略括号还有一个条件 —— 没有可选说明符。如果这里我们给lambda表达式添加一个`mutable`说明符，那么编译器可能会报告一个警告或者错误。

```
auto foo = [y] mutable { return x + y + z; };
```

在新版本的编译器中，通常会让代码编译通过，并且给出警告提示需要C++23标准。当然如果编译器版本比较老，这里会直接给出编译错误。

clang 13的警告：

```
warning: lambda without a parameter clause is a C++2b extension [-Wc++2b-
extensions]
```

clang 12的报错：

```
error: lambda requires '()' before 'mutable'
```

C++23标准修改了上述的限制，是否可以忽略括号不再受到说明符的影响，即编译上述代码不再有任何警告和错误提示了。

17. 强制的类成员声明顺序布局

在C++23标准之前，如果类的非静态数据成员具有不同的访问控制权，那么标准是允许编译器对这些数据成员做重新排布的。也就是说，一旦发生重新排布，那么声明的顺序和内存布局的顺序就不同了。

不过，据调查主流的一些编译器包括MSVC、Clang和gcc都没有此进行实现，也就是说真实布局和声明顺序一样。很显然，声明顺序和内存布局相同更容易让程序员理解，而且主流编译器都是这样实现的，可谓是好处颇多。

所以C++23标准取消了允许编译器对具有不同访问控制权的非静态数据成员进行重新排布的规则。值得一提的是，这个修改并不影响标准布局的判定`std::is_standard_layout`的判断结果不会发生改变，所以不必担心兼容性问题。

18. 多维下标运算符

为了更好的支持多维容器和视图，C++23标准引入了多维下标运算符特性，虽然该特性对于C++标准来说是新的，但是读者朋友应该早就在使用其他语言时接触过它。例如C#的多维数组：

```
int [,] my_data = new int [3,3] {
    {1, 2, 3}, {4, 5, 6}, {7, 8, 9}
};
int elem = my_data[1, 1];
```

以及python的NumPy数组（`numpy.array`）：

```
import numpy as np
my_data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
elem = my_data[1, 1]
```

这里`my_data[1, 1]`就是通过多维下标运算符访问了第二行第二列的元素。

在C++23标准之前，C++的下标运算符函数只接受一个参数，如果要访问多维数组的第二行第二列的元素，就需要使用单参数下标运算符链的形式：

```
int elem = my_data[1][1];
```

很显然，用上述这种方式去访问多维数据类型多少有点脱离时代了。在矩阵、张量和图形图像处理大行其道的今天，C++需要一种方法能够更加方便的访问多维数据类型。于是，多维下标运算符被引入到C++23标准中。标准修改了下标运算符`operator[]`的定义，让下标运算符`operator[]`支持零个或多个参数，包括可变参数。它的用法和定义都与函数调用运算符`operator()`相匹配。

注意，这里会牵扯到一个兼容问题。还是以

```
int elem = my_data[1, 1];
```

为例，在C++20标准以前，`1, 1`是一个合法的逗号表达式，所以这段代码实际意义为：

```
int elem = my_data[1];
```

C++20标准明确弃用了在下标运算符顶层使用逗号表达式的用法（见34.11节），使用逗号表达式会收获编译器的警告：

```
warning: top-level comma expression in array subscript is deprecated in C++20
```

到了C++23标准，上述写法从语法上说已经是一种格式错误了。当然，不同的实现对其的宽容程度还是不同的，例如GCC会让代码编译成功，只是给出警告：

```
warning: top-level comma expression in array subscript changed meaning in C++23
```

而Clang会直接报错：

```
error: type 'int[5]' does not provide a subscript operator
```

虽然这个兼容问题无法避免，但是读者朋友也不必过于担心，因为逗号表达式出现在下标运算符内部的情况非常罕见，并且如果执意要使用逗号运算符，可以用一对括号将其包围：

```
int elem = my_data[(1, 1)];
```

读到这里，有朋友可能会提出疑问，在C++23标准之前，为了更方便的访问多维数据结构对象的元素，聪明的C++程序员也找到了一些方法，例如使用函数调用运算符访问元素：

```
int elem = my_data(1, 1);
```

或者使用元组或者类似元组的数据对象来访问元素：

```
int elem = my_data{1, 1};
```

又或者干脆使用传统数组访问语法，并不觉得此方案有问题：

```
int elem = my_data[1][1];
```

这些方法都可以满足需求，为何还要引入新的特性并且造成一定的兼容性问题呢？

答案是需要引入新特性，上述两种方案也有其缺陷。

1. 对于使用函数调用运算符访问元素的方案，首先我们会发现数组访问和对象的访问不一致，这必然会让新手C++程序员感到困惑，尤其是看到赋值左侧似乎是函数调用的时候，例如`my_data(1, 1) = 42;`。其次，使用`operator[]`比`operator()`从语义上更加让人一目了然，符合大多数编程语言的设定。最后，区分函数调用运算符和下标运算符更容易在遇到错误的时候进行排查，不容易被错误提示混淆。
2. 对于使用元组或者类似元组的数据对象来访问元素，其存在的一个问题和上述第一个问题相同，数组访问和对象的访问不一致。另外一个问题是，每次访问数据之前需要先构造一个新的对象，不利于代码的优化。
3. 对于使用传统数组访问语法来访问元素，首先，`a[x][y]`意味着`a[x]`是一个有效的表达式，对于任何自定义数组类型，这意味着`a[x]`是一个代理引用。这使得数组访问所需的函数调用深度不小于维数，这阻碍了内联优化，编译器不能内联数组访问会对执行效率产生不良影响。如果这种数组访问的次数不多，那么可能看不出存在效率问题，一旦出现大量数组访问的情况，程序将面临严重的效率问题。其次，获取`a[x]`付出的代价可能远高于直接访问`a[x][y]`，例如，对于基于压缩列

的稀疏矩阵，`a[x]` 将表示整行的视图。构建这个视图要比仅仅获取行 `x` 和列 `y` 处的元素复杂得多。对于这种稀疏矩阵格式，列导向的访问速度比行导向的访问更快，因此更常见的是获取列视图而不是行视图。第三，`a[x][y]` 不会告知 `a[x]` 的拷贝行为，例如对于 `std::vector<std::vector<int>>` 它是一个深拷贝，而对于 `int**` 它是一个浅拷贝。第四，`a[x, y, z]` 比 `a[x][y][z]` 更加容易扩展，因为前者可以支持包展开，可以通过包展开的特性支持任意维度的对象。

现在，让我们通过一个提案文档中的例子来领略多维下标运算符的实用之处：

```
#include <cassert>
#include <array>
#include <span>
template <class From, class To>
concept convertible_to =
    std::is_convertible_v<From, To> &&
    requires(std::add_rvalue_reference_t<From> (&f)()) {
        static_cast<To>(f());
    };
template <typename T, std::size_t S>
struct array;
template <typename T, auto N = 0>
constexpr inline bool _is_array = false;
template <typename T, auto N>
constexpr inline bool _is_array<array<T, N>> = true;

template <typename T, std::size_t S>
struct array : std::array<T, S>
{
    static constexpr inline std::size_t extent = []() -> std::size_t
    {
        if constexpr (_is_array<T>)
        {
            return 1 + T::extent;
        }
        return 1;
    }();
    constexpr decltype(auto)
    operator[](std::size_t idx)
    {
        return *(this->data() + idx);
    }
    constexpr decltype(auto)
    operator[](std::size_t idx, convertible_to<std::size_t> auto &&...args)
        requires(sizeof...(args) < extent) && (sizeof...(args) >= 1)
    {
        typename std::array<T, S>::reference v = *(this->data() + idx);
        return v.operator[](args...);
    }
    constexpr decltype(auto)
    operator[](std::size_t idx) const
    {
        return *(this->data() + idx);
    }
    constexpr decltype(auto)
    operator[](std::size_t idx, convertible_to<std::size_t> auto &&...args) const
```

```
        requires(sizeof...(args) < extent) && (sizeof...(args) >= 1)
    {
        typename std::array<T, S>::reference v = *(this->data() + idx);
        return v.operator[](args...);
    }
};

template<class T, class... U>
array(T, U...) -> array<T, 1 + sizeof...(U)>;

int main() {
    // 2 x 3 二维数组
    array aa{array{1, 2, 3}, array{4, 5, 6}};
    static_assert(decltype(aa)::extent == 2);
    assert((aa[0, 1] == 2)); // 多余的括号是因为assert是宏
    assert((aa[1, 2] == 6));
    array bb{array{7, 8, 9}, array{10, 11, 12}};
    array cc{array{13, 14, 15}, array{16, 17, 18}};
    array dd{array{19, 20, 21}, array{22, 23, 24}};
    // 4 x 2 x 3 三维数组
    array aaa{aa, bb, cc, dd};
    static_assert(decltype(aaa)::extent == 3);
    assert((aaa[1, 1, 1] == 11));
}
```

这份代码比较复杂，需要掌握模板推导指引、`decltype(auto)`推导规则、常量表达式函数、可变模板参数、包展开以及概念和约束等特性才能完全理解，这里解释与多维下标运算符相关的核心部分。

`array`实现了两对多维下标运算符函数，分别对应常量对象和非常量对象，因为原理完全相同，所以我们只看非常量对象对应的多维下标运算符函数。

```
constexpr decltype(auto)
    operator[](std::size_t idx)
```

和

```
constexpr decltype(auto)
    operator[](std::size_t idx, convertible_to<std::size_t> auto &&...args)
        requires(sizeof...(args) < extent) && (sizeof...(args) >= 1)
```

是多维下标运算符的两个重载函数，分别有1个参数和可变参数。当参数为1的时候，代码很简单，只是返回一维数组中对应的元素。当参数大于1的时候，情况稍微复杂一些。首先，当参数大于1时，多维下标运算符函数的参数用到了包展开，编译器会自动将`args`展开为调用者指定的实参。其次，这里定义了两个约束，一方面约束`args`的类型必须可以转换为`std::size_t`，另一方面，约束了`args`的个数必须大于等于1，小于类型的维度。在确保满足上述条件之后，函数通过`idx`获取外层数组数据的引用，由于引用的还是`array`类型，于是可以通过包展开`args...`继续调用类型的多维下标运算符函数。若`args...`展开后是多个实参，则调用有可变参数的多维下标运算符的重载版本，反之则调用单参数的多维下标运算符的重载版本。

值得一提的是，代码中`array{1, 2, 3}`可以成功构造`array`对象不是因为类模板的模板实参推导，`array`是一个典型的聚合类型，没有构造函数。能够成功构造是因为模板推导指引：

```
template<class T, class... U>
array(T, U...) -> array<T, 1 + sizeof...(U)>;
```

19. 具名通用字符转义

通用字符转义的引入让我们可以很方便的定义不容易输入的字符，比如定义希腊大写字母Δ：

```
char delta[] = "\u0394";
```

不过，不知道大家有没有注意到，使用这种方法来定义字符会让代码中出现意义不明的代码，如`0394`，虽然我们知道它是一个Unicode字符的代码值，但是具体是哪一个却是这个值无法直接表达清楚的，需要添加注释或者在变量命名上加以说明才行。

为了更加优雅的表示一个Unicode字符，C++23标准引入了具名通用字符转义的特性，也就是说我们可以使用Unicode字符的名称来定义字符了，还是以定义希腊大写字母Δ为例，我们可以将上述代码改写为：

```
char delta[] = "\N{GREEK CAPITAL LETTER DELTA}";
```

注意，这里的语法与分隔的转义序列相似，其中`\N`是转义引入符，后面紧跟的大括号中的字符序列是字符名称或者字符别名。其具体语法要求为：

1. 字符序列必须仅包含大写字母A到Z、数字、空格和连字符减号；
2. 字符序列必须精确匹配Unicode 分配的名称（与 ISO/IEC 10646 同步）或者
3. 必须精确匹配Unicode 别名（与 ISO/IEC 10646 同步），而且别名类型必须是“control”，“correction” 或 “alternate” 。

> 值得注意的是标准并没有支持Unicode 命名序列（与 ISO/IEC 10646 同步）

熟悉Python的读者朋友肯定会发现上述语法和Python中具名字符转义的语法是高度相似的，因为该提案文档明确说明参考了Python的语法。

经常使用UTF-8作为源代码编码的朋友可能会对该特性提出一个疑问：使用UTF-8编码的源文件完全可以在字符串中显式的输入Unicode字符，并不需要使用字符转义，例如：

```
char delta[] = "Δ";
```

那么该特性是否真的有必要引入到C++标准中呢？答案是有必要，C++委员会的专家针对这个问题给出了两个原因：

1. 并非所有平台的源代码都默认采用UTF-8编码，比如Windows环境MSVC仍然默认使用与区域设置相关的编码，并鼓励将源文件限制为ASCII编码。
2. 即使有些Unicode字符可以直接写在源代码中，也会造成源代码的混乱。比如不可见的字符`U+200B {ZERO WIDTH SPACE}`，组合字符`U+0300 {COMBINING GRAVE ACCENT}`，视觉上不容易区分的字符`U+003B {SEMICOLON}`和`U+037E {GREEK QUESTION MARK}`。

字符	代码
""	U+200B {ZERO WIDTH SPACE}
"̀"	U+0300 {COMBINING GRAVE ACCENT}
";"	U+003B {SEMICOLON}

字符	代码
";"	U+037E {GREEK QUESTION MARK}

很显然，在这种情况下，使用转义序列可以提高代码清晰度。因此，使用具名通用字符转义序列的意义仍然存在。

20. 明确`static_assert`和`if constexpr`支持`bool`缩窄转换

C++23标准明确了`static_assert`和`if constexpr`可以进行`bool`缩窄转换。虽然主流编译器绝大多数情况都支持它们的`bool`缩窄转换，但是在此之前并没有标准对其做规定。例如以下代码：

```
template <std::size_t N>
class Array
{
  static_assert(N, "no 0-size Arrays"); // 需要缩窄转换
  // ...
};

Array<16> a;
```

上面的代码无论使用MSVC、GCC还是Clang编译都可以编译通过，但这不是标准规定的。`if constexpr`的情况也是类似的：

```
enum Flags { Write = 1, Read = 2, Exec = 4 };

template <Flags flags>
int f() {
    if constexpr (flags & Flags::Exec) // 需要缩窄转换
        return 0;
    else
        return 1;
}

int main() {
    return f<Flags::Exec>();
}
```

MSVC、GCC都可以编译通过。Clang比较特殊，Clang12以及之前的版本会编译报错，提示为了目标类型无法缩窄转换为`bool`类型。不过从Clang13开始，这段代码可以编译成功，甚至不会出现警告。

实际上，让`static_assert`和`if constexpr`支持缩窄转换到`bool`更符合一般语法，因为与之对应的运行时代码是没有问题的：

```
if (flags & Flags::Exec)
{
    // ...
}

assert(N);
```

为了解决上述问题，C++23标准修改了部分措辞，包括：

1. 对于`static_assert`，将参数部分的描述由“类型为 `bool` 的上下文转换常量表达式”，修改为“根据上下文转换为 `bool` ，且转换后的表达式应为常量表达式”。
2. 对于`if constexpr`，将条件部分的描述由“应为 `bool` 类型的上下文转换常量表达式”，修改为“根据上下文转换为 `bool` 且转换后的表达式应为常量表达式”。

经过修改后，下面的代码均符合标准规范：

```
static_assert(sizeof(int[2]));
if constexpr (sizeof(int[2])) {}
```

21. 允许非字面量变量和`goto`语句的常量表达式函数

我们已经知道了，从C++14到C++20，标准对常量表达式函数进行了很大程度上的增强。在这些新标准的加持下，编写常量表达式函数已经基本上没有什么特殊之处，除了不能声明非字面量变量（包括静态变量和线程局部存储变量）和`goto`语句。但我们还是会发现一个问题，无论是非字面量变量还是`goto`语句都不会在一个非常量求值的路径下造成不良影响，而且恰好相反，对于常量表达式函数中非常量求值的路径，增加对非字面量变量和`goto`语句是非常简单并且实用的一件事情。

让我们看看下面这个例子：

```
#include <type_traits>
constexpr int f(int i)
{
    if (std::is_constant_evaluated()) {
        return i;
    }
    else {
        static int var = 0;
        return i * ++var;
    }
}

int main()
{
    constexpr int var = f(7);
    int var2 = f(11);
}
```

上面的代码使用C++20标准是不应该编译成功的，这里描述为“不应该”而不是“不能”，因为新版的Clang16编译器可以编译成功，只是会给出一条警告，告知我们应该使用新的C++标准来编译这份代码。这份代码不应该编译成功的原因是在常量表达式函数中的`else`分支中存在一个静态变量，这是C++23之前的标准所不允许的。

很明显，这种规定非常的不合理。首先，作为常量求值的`f(7)`，它的函数执行路径不会经过静态变量，所以在编译期求值的时候完全不会考虑静态变量的影响；其次，在运行时调用`f(11)`，在`else`的分支中完全不需要进行常量求值，这种情况下不允许静态变量存在的规则是多余的。同样的，`goto`语句也会有这样的情况：

```
#include <type_traits>
constexpr int f(int i)
{
    if (std::is_constant_evaluated()) {
        return i;
    }
    else {
        goto L;          // C++23标准之前会编译报错
        i++;
L:
        return i;
    }
}

int main()
{
    constexpr int var = f(7);
    int var2 = f(11);
}
```

为了解决上述问题，C++23标准规定允许非字面量变量（包括静态变量和线程局部存储变量）和`goto`语句在常量表达式函数中声明，前提条件是它们不能出现在常量求值的编译路径。因为上述两份代码的静态变量和`goto`语句都没有经过常量求值的控制流，所以它们在C++23标准的环境中可以编译成功。

但是，如果我们的代码没有遵守这个前提条件，那么编译器还是会报错，例如下面的代码：

```
#include <type_traits>
constexpr int f(int i)
{
    static int var1 = 0;                // 编译错误：常量求值的控制流经过了静态变量
    if (std::is_constant_evaluated()) {
        thread_local int var2 = 0;      // 编译错误：常量求值的控制流经过了线程局部存储变
量
        return i;
    }
    else {
        return i;
    }
}

int main()
{
    constexpr int var = f(7);
    int var2 = f(11);
}
```

最后值得一提的是，常量表达式规则的持续修改也表明了C++的一个发展方向，即尽量不在声明或者定义时而是在使用时诊断语言结构，对于没有使用的代码，我们不对其进行诊断。# 允许常量表达式函数中声明static constexpr变量

我们在前面的章节中已经看到了，C++23标准规定如果代码执行的控制流不经过静态变量，那么常量表达式函数就能够顺利编译。不过仔细想想，这里其实还存在一个问题，难道所有的静态变量都会影响到常量表达式函数在编译阶段求值么？显然不是的，比如下面这个例子：

```
char xdigit(int n)
{
    static constexpr char digits[] = "0123456789abcdef";
    return digits[n];
}
```

在这份代码中`digits`是一个常量表达式变量，无论什么时候它都不会影响到编译器在编译阶段求值。作为一个普通函数`xdigit`可以顺利编译通过，但是如果将其声明为`constexpr`情况就不同了：

```
constexpr char xdigit(int n)
{
    static constexpr char digits[] = "0123456789abcdef";
    return digits[n];
}
```

上面这份代码无法编译成功，编译器给出的理由仅仅是`static`出现在了常量表达式函数中，但是我们刚刚也讨论过，事实上`static constexpr char digits[]`根本不会对编译阶段的常量求值有任何影响，因为根据定义对于`static constexpr`变量来说，它必须是常量初始化的，所以这个限制是非常没有必要的。

于是C++23标准对于这个限制做出了修改，它规定如果变量可以在常量表达式中使用，那么它就可以在常量表达式函数中进行声明。在做出了这个修改之后，`constexpr`版本的`xdigit`函数就可以顺利编译通过了。

最后值得一提的是，我们可以注意到标准的措辞是“变量可以在常量表达式中使用”，并非规定必须是`static constexpr`声明的变量，所以下面这份代码使用C++23标准也可以编译通过：

```
constexpr int get_number()
{
    static const int retval = 42;
    return retval;
}
```

22. 进一步放宽常量表达式函数的限制

C++23标准进一步放宽了常量表达式函数的限制，简单来说就是现在可以编写一个永远无法满足编译阶段进行常量求值的常量表达式函数，并且这个常量表达式函数可以编译成功。

例如以下代码：

```
void f(int& i)
{
    i = 0;
}

constexpr void g(int& i)
{
    f(i);
}
```

请注意，这份代码在C++23标准之前是无法编译成功的，因为函数`g`无条件的调用了函数`f`，并且函数`f`不是常量表达式函数，不存在常量求值的可能，所以会编译报错。注意，这段代码GCC和MSVC会编译报错，Clang需要选择Clang11以及之前的版本，否则是可以编译成功的。

如果只是参考上述这个例子，也许有的朋友会认为C++23标准的这个新规定有一些多此一举。因为函数`g`的`constexpr`声明毫无意义，它不可能是常量表达式函数，所以编译器指出这个错误可以帮助程序员优化代码。

这是一个很有说服力的说法，但是考虑下面的例子，情况可能就不同了：

```
#include <optional>

constexpr void h(std::optional<int>& o)
{
    o.reset();
}
```

在这个例子中，函数`h`和函数`g`执行的内容并没有太大区别，都是调用一个没有声明为`constexpr`的函数。当然，这里的"没有声明为`constexpr`的函数"需要进一步解释，在C++17标准中，`reset`是没有声明`constexpr`的，所以必然会导致编译报错。但是情况在C++20和C++23标准发生了改变，C++20中`reset`是否声明为`constexpr`取决于标准库的实现，而C++23标准规定`reset`为常量表达式函数。那么这里的问题是，标准库的代码很容易的将函数声明为了`constexpr`的版本，为了适配这个函数的变化，我们也需要做出相对应的修改来获取更好的性能：

```
#include <optional>

#if __cpp_lib_optional >= 202106
constexpr
#endif
void h(std::optional<int>& o)
{
    o.reset();
}
```

如果是明确的，有办法辨识出修改边界的标准库函数，我们是可以按照上面的方式来处理函数`h`，但是面临的两个困难是，首先标准库函数众多，我们必须有足够的精力去跟踪标准N和标准N+1中关于`constexpr`函数的变化。其次，如果我们使用的是非标准库，那么适配常量表达式函数的任务将会更加艰巨。

提案文档中坦诚的提到，关于常量表达式函数的这个限制在C++11标准的时候是有意义的，但是到了C++23标准，常量表达式函数越来越宽松，使用常量表达式函数的代码库也越来越多，那么对于常量表达式函数最友好的处理方式其实是将诊断常量表达式的任务保留到我们必须诊断它的地方，即那些编写的必须在编译期常量求值的代码：

```
int f(int& i)
{
    i = 0;
    return i;
}

constexpr int g(int& i)
{
    return f(i);
```

```
}

int main()
{
    int j = 0;
    auto x = g(j);                // 编译成功，没有进行常量求值
    constexpr auto y = g(j);      // 编译失败，g(int& i)无法进行常量求值
}
```

上面这段代码中，`auto x = g(j);`可以编译成功，因为代码没有要求必须进行常量求值，而`constexpr auto y = g(j);`因为代码要求变量`y`在编译期间完成计算，但是`g(int& i)`无法满足这个要求，所以编译失败。

C++23标准放宽常量表达式函数的限制不仅体现在用户定义函数，而且还包括编译器生成的函数，例如下面的代码：

```
template <typename T>
struct Wrapper {
    constexpr Wrapper() = default;
    constexpr Wrapper(Wrapper const&) = default;
    constexpr Wrapper(T const& t) : t(t) { }

    constexpr T get() const { return t; }
    constexpr bool operator==(Wrapper const&) const = default;
private:
    T t;
};

struct X {
    X();
    bool operator==(X const&) const;
};

Wrapper<X> x;
```

编译这份代码，GCC会报错：

```
error: call to non-'constexpr' function 'bool X::operator==(const X&) const'
note: 'bool X::operator==(const X&) const' declared here
```

错误的原因和上一个例子一样，是`constexpr bool operator==(Wrapper const&) const`会调用了非`constexpr`函数`bool X::operator==(const X&) const`，即使`Wrapper<X> x;`这句代码根本不会调用该函数，这显然也是不合理的。使用C++23标准，上述代码可以顺利的编译通过，即使该函数是一个编译器生成的显式默认函数。

回顾一下`constexpr`的发展进程，常量表达式函数的限制几乎在每个发布的标准中都会放宽，比如本节特性的提案文档，该提案文档对于标准的措辞进行了大量的删减，目的就是减少`constexpr`的限制。通过这些标准的变化我们会发现，C++委员会在C++发展道路上拟定的一个目标，即不在声明或者定义时对代码进行诊断，而是保留到真正调用时再做。

23. 禁止混合字符串字面量的连接

在介绍新字符类型的章节中，我曾提到过关于字符串连接的规则，让我们回顾一下：如果两个字符串字面量具有相同的前缀，则生成的连接字符串字面量也具有该前缀。如果其中一个字符串字面量没有前缀，则将其视为与另一个字符串字面量具有相同前缀的字符串字面量，**其他的连接行为由具体实现者定义。**

请注意这句加粗的解释，“其他的连接行为由具体实现者定义”，这句话的意思是混合字符串字面量的连接结果由编译器自己决定，标准并不干涉最终结果。请注意这个规则是C++11标准就提出来的，但是无论是GCC、Clang还是MSVC都没有实现混合字符串字面量的连接，简单来说就是编译器会针对混合字符串字面量的连接报告编译错误，例如：

```
void f() {

  { auto a = L"" u""; }
  { auto a = L"" u8""; }
  { auto a = L"" U""; }

  { auto a = u8"" L""; }
  { auto a = u8"" u""; }
  { auto a = u8"" U""; }

  { auto a = u"" L""; }
  { auto a = u"" u8""; }
  { auto a = u"" U""; }

  { auto a = U"" L""; }
  { auto a = U"" u""; }
  { auto a = U"" u8""; }
}
```

上面的代码使用三大编译器都会编译失败，其中GCC的错误提示为：

```
error: concatenation of string literals with conflicting encoding prefixes
```

MSVC和Clang也有类似错误提示。

提案认为这种混合字符串字面量的连接到目前看来是没有意义的，所以C++23标准规定混合字符串字面量的连接是不正确的。

24. 删除不可编码的宽字符和多字宽字符字面量

在C++23标准以前，宽字符类型是可以被超出类型长度的宽字符字面量赋值，甚至是可以被多个宽字符字面量赋值的。例如下面的代码：

```
wchar_t a = L'🤦';  // \U0001f926
wchar_t b = L'ab';  // \u0061\u0062
wchar_t c = L'é';   // \u00e9       NFC - 标准等价合成
wchar_t d = L'é';   // \u0065\u0301 NFD - 标准等价分解
```

这份代码看似很简单，是4条赋值语句，但实际上这里面包含了很多问题，我来详细解释一下：

1. 变量`a`被赋值为`L'🤦'`，其中`L'🤦'`的编码为`0x0001f926`。请注意，由于`wchar_t`类型在不同的环境下类型长度是不同的，所以变量`a`的实际状态会有比较大的差别。例如在linux中，`wchar_t`类型长度为32位，也就是4个字节，那么它的长度足以容纳`L'🤦'`的编码，不会出现问题。但是如果是在Windows这样`wchar_t`类型长度为16位的环境中，情况就不太好了。由于长度只有2个字节，所以`L'🤦'`的编码会被截断，截断的方式是先转换为UTF-16，然后只获取前16位的数据`0xd83e`，显然转换后的数据已经无法表达字符的原本含义了。另外，如果用GCC结合`-fshort-wchar`参数选项，在linux下让`wchar_t`转换到16位，其行为还是会跟Windows有所区别，虽然GCC也会将字符转换为UTF-16，但是取的是后16位的数据`0xdd26`。可以看到，`wchar_t`为16位的情况下，无论我们怎么做，原字符字面量的数据都无法保留，并且Linux和Windows会得到不同的结果。
2. 对于多字宽字符也会存在类似问题，上面的代码中变量`a`被赋值为`L'ab'`。对于MSVC而言，它会取首个字符，也就是`a`，而对于GCC和Clang，它们会选择最后的字符，也就是`b`，也存在跨平台的兼容性问题。
3. 最后，也是最有趣的部分，变量`c`和变量`d`分别被赋值为`L'é'`和`L'é'`。请注意，这里的`L'é'`和`L'é'`是不同的。使用标准等价合成(NFC)编码字符，`L'é'`的编码为`0x00e9`，MSVC、GCC和Clang不会出现问题。但是采取标准等价分解(NFD)编码字符，`L'é'`的编码为`0x0065`和`0x0301`，这个时候又会出现上述的分歧，MSVC获取的编码为`0x0065`，而GCC和Clang的编码为`0x0301`。因此，有时候源代码中看似相同的单字符字面量，实际上可能是多字符字面量，导致兼容性问题。
4. 值得注意的是这份代码在不指定C++23标准的情况下，GCC和MSVC都是可以编译成功的，当然在编译过程中会给出几条警告信息。Clang比较特殊Clang14以及以上版本会直接编译失败，需要Clang13版本才能编译成功。

通过上面的解释，我们可以看出，编译器在处理多字宽字符字面量的时候有着非常大的区别，导致兼容性问题，这个是需要C++23标准来解决的。

因此，C++23标准规定不可编码的宽字符和多字宽字符字面量是不正确的。因为这种形式的宽字面量造成了编译器实现的分歧，并且没有实现可以在不丢失信息的情况下处理好不可编码的宽字符和多字宽字符字面量，影响了代码的可移植性。

> 最后介绍下怎么在计算机中简单的输入上述的一些特殊字符吧。如果没有掌握输入这些字符的方法，想必做起实验也不会顺利。
>
> 我这里推荐使用Chrome或者Edge的开发工具的控制台，使用console.log来输出特殊字符后复制到源代码，比如输出字符🤦，我们只需要输入：
>
> console.log('\u{1f926}')
>
> 再比如输出é：
>
> console.log('\u0065\u0301')

25. 可选的扩展浮点类型

近年来，随着AI的兴起，浮点计算对计算机世界的重要性已经提到了前所未有的高度。硬件和软件为了更好的支持浮点计算都投入了大量的精力。以16位的浮点类型为例，在ARM CPU、NVIDIA GPU 以及Intel CPU等硬件都进行了支持，软件方面OpenGL、CUDA 和 LLVM IR等也能支持16位浮点类型。但是在C++23标准之前，C++程序员很难使用16位浮点类型，因为编译器缺乏对该类型的内置支持。常见的解决方法是定义一个包含所有转换运算符和重载算术运算符的类类型，以使其行为尽可能类似于内置类型。但这种方法既麻烦又不完整，需要内联汇编或其他特定于编译器的技巧才能生成高效的代码。

缺少完善的浮点类型支持不仅仅存在与C++，C语言也有同样的问题。所以，C23和C++23标准都对浮点类型进行了可选的扩展。C语言标准使用 `_Float16` 、 `_Float32` 、 `_Float64` 和 `_Float128` 作为命名 IEEE 类型的可选关键字，而C++使用 `std::float16_t` 、 `std::float32_t` 、 `std::float64_t` 和 `std::float128_t` 类型别名，它们定义在头文件 `<stdfloat>` 中。

从上面的描述看来，虽然C和C++在扩展浮点类型名方面有一些分歧，但是在编译器实现上一般会选择保持兼容，即使编译器没有这么做，也不是什么问题，我们要做的只是多些几行代码而已：

```
#ifdef __cplusplus
  #include <stdfloat>
  using my_fp16_t = std::float16_t;
#else
  typedef _Float16 my_fp16_t;
#endif
```

另一个值得注意的区别是缩窄转换，C++23标准中扩展浮点类型之间的缩窄转换必须是显式的，而C23标准扩展浮点类型之间的转换可以隐式完成，即使这种转换会损失数据。因此代码能使用C语言编译成功，可能会在C++上编译失败。不过问题也不大，把代码改为显式转换即可，这样C和C++都能编译成功了。

注意，扩展浮点类型是否支持无穷大、NaN以及指数的底数是多少，依旧是编译器实现定义，标准没有改变。

除了上述C和C++都具备的扩展浮点类型，C++标准还定义了一个 `std::bfloat16_t` 类型，该类型用于Google的TPU和TensorFlow，并在NVIDIA最新的GPU中获得硬件支持。该类型的特点是使用8位指数位来保留32位浮点数 `std::float32_t` 的近似动态范围，但它仅支持8位精度。简单来说，由于精度限制，它不合适做整数多的计算，但它减少存储需求，这就对机器学习算法非常友好了。

与扩展浮点类型一起添加的还有内置字面量后缀 `std::float16_t` 、 `std::float32_t` 、 `std::float64_t` 、 `std::float128_t` 和 `std::bfloat16_t` ，它们分别对应 `f16` 、 `f32` 、 `f64` 、 `f128` 和 `bf16` 以及大写版本的 `F16` 、 `F32` 、 `F64` 、 `F128` 和 `BF16` 。没有考虑使用标准库实现字面量后缀的原因是，需要 `#include` 和 `using namespace` 指令才能使用它们，不利于和C语言的兼容性。

下面的表格展示了各个扩展类型的具体信息：

C++类型名	C类型名	字面量后缀	类型长度	精度位数	指数位数
std::float16_t	_Float16	f16 或 F16	16	11	5
std::float32_t	_Float32	f32 或 F32	32	24	8
std::float64_t	_Float64	f64 或 F64	64	53	11
std::float128_t	_Float128	f128 或 F128	128	113	15
std::bfloat16_t	无	bf16 或 BF16	16	8	8

最后，对于扩展浮点类型，还有4个需要注意的地方：

1. 关于隐式转换标准规定，当至少其中一种类型是扩展浮点类型时，仅当目标类型的转换等级大于或等于源类型时，两个浮点类型之间的转换才是隐式的。任何隐式转换都是无损的并准确保留值，任何可能有损的转换都必须是显式的。

```
#include <stdfloat>

float f32v = 1.0;
std::float16_t f16v = 2.0f16;
std::bfloat16_t b16v = 3.0bf16;
auto x = f32v + f16v; // 编译成功，float等级大于std::float16_t，f16v转换位float类型
auto y = f32v + b16v; // 编译成功，float等级大于std::bfloat16_t，b16v转换位float类型
auto z = f16v + b16v; // 编译失败，两种类型都不能通过算术转换转换为另一种类型
```

2. 由于C++标准规定缩窄转换需要显式进行，所以标准浮点类型到扩展浮点类型的转换也是如此，这里的问题是浮点数字面量是`double`类型，例如：

```
std::float16_t f16v = 1.0;
```

浮点数`1.0`是`double`类型，因此赋值操作发生了缩窄转换，标准是不支持这样做的，目前GCC13会给出编译警告。

这里正确的做法是：

```
std::float16_t f16v = static_cast<std::float16_t>(1.0);
```

或者

```
std::float16_t f16v = 1.0f16;
```

3. 如果转换等级相同，标准浮点类型和扩展浮点类型是扩展浮点类型，这一点和C语言保持一致：

```
#include <stdfloat>
#include <type_traits>

float f32a = 1.0;
std::float32_t f32b = 2.0f32;
auto x = f32a + f32b;
static_assert(std::is_same_v<decltype(x), std::float32_t>);
```

4. 关于函数重载中存在扩展浮点类型转换的情况，优先选择具有相同转换等级的浮点类型：

```
#include <stdfloat>

void f(std::float32_t);
void f(std::float64_t);

int main()
{
    f(std::float16_t(1.0)); // 编译失败，二义性问题
    f(float(2.0));          // 调用std::float32_t版本，转换等级相同
    f(double(3.0));         // 调用std::float64_t版本，唯一可用
}
```

在第一个调用 `f(std::float16_t(1.0))` 中，两个重载都是可行的。但由于所涉及的类型 `std::float16_t` 、 `std::float32_t` 和 `std::float64_t` 都没有相同的转换等级，因此上述浮点类型的特殊规则不适用，存在二义性问题，导致编译失败。

而在调用 `f(float(2.0))` 中，两个重载也都是可行的，本来也应该触发二义性问题，但根据标准针对浮点类型的特殊规则，从 `float` 到 `std::float32_t` 的转换优先于从 `float` 到 `std::float64_t` ，因为 `float` 和 `std::float32_t` 具有相同的转换等级，所以可以编译成功。

26. 允许static_asserts参数与if constexpr条件语句缩窄转换到bool类型

在C++23标准之前， `static_asserts` 参数和 `if constexpr` 的条件的描述是：

> shall be a contextually converted constant expression of type bool

简单来说就是一个应为 `bool` 类型的常量表达式，而从在C++23标准开始，标准中的描述修改为：

> is contextually converted to bool and the converted expression shall be a constant expression

翻译过来是，能够根据上下文转换为 `bool` 并且转换后的表达式应为常量表达式。很明显，这里的修改强调了“可转换为 `bool` ”。

具体情况以下代码：

```
enum Flags { Write = 1, Read = 2, Exec = 4 };

template <Flags flags>
int f() {
    if constexpr (flags & Flags::Exec) // C++23标准之前编译错误，无法缩窄转换
        return 0;
    else
        return 1;
}

template <unsigned int N>
class Array
{
    static_assert(N, "no 0-size Arrays"); // C++23标准之前编译错误，无法缩窄转换
};

int main() {
    Array<16> a;
    return f<Flags::Exec>();
}
```

在这份代码中， `static_assert` 的参数和 `if constexpr` 的条件表达式都不是 `bool` 类型，按照C++23之前的标准是无法编译通过的。不过幸运的是，问题并没有想的那么严重，因为编译器的实现可以和标准是有所出入的。例如 `if constexpr` 语句，这份代码使用GCC编译，新老版本的编译器都能够顺利的编译通过，没有出现编译错误。而使用Clang进行编译，也只有Clang13之前的编译器会报错，新的Clang编译器也可以顺利编译通过。另外MSVC编译器，最多只是给出警告，并不会影响编译结果。至于 `static_assert` 的参数，三大编译器更是非常默契的选择允许缩窄到 `bool` 的转换，所以无论使用新老版本的编译器都可以顺利的通过编译。

顺带一提，当常量的值是1或者0的整数时，转换到`bool`的过程不是缩窄转换：

```
const int x = 0;
void f() noexcept(sizeof(char[2])) {}
void g() noexcept(sizeof(char)) {}
void h() noexcept(x) {}
```

上面的代码中，`g()`和`h()`两个函数都可以在Clang中编译成功，因为`noexcept`的实参为常量0和1，所以没有发生缩窄转换。而函数`f()`则会编译报错，因为常数2转换为`bool`类型是缩窄转换，而缩窄转换在noexcept中是不允许的，所以最终导致编译错误。

```
error: noexcept specifier argument evaluates to 2, which cannot be narrowed to
type 'bool'
```

27. 静态下标运算符函数

无独有偶，C++23标准除了允许函数调用运算符函数声明为`static`，下标运算符函数也被允许声明为`static`，也就是说下面这份代码的语法在C++23标准中是符合规范的：

```
struct X {
    static bool operator[](int);    // C++23标准可以成功编译
    static bool operator()(int);    // C++23标准可以成功编译
};
```

关于为什么要允许下标运算符函数声明为`static`，C++委员会给出的答案很简单，因为我们应该尽可能的让成员函数`operator[]`的行为接近`operator()`。在我看来这个理由也算是足够的充分了，毕竟这个特性的影响范围也非常的小。

28. 支持UTF-8作为可移植源文件编码

在C++23标准以前编译器支持的源文件字符编码完全是由编译器实现自定义的，也就是说，即使我们写了一行最简单的代码并保存为文件：

```
int main() {}
```

也有可能编译失败，因为文件的字符编码可能在某些编译器上并不支持。如果我们想编写的代码在所有的编译器上都能够编译成功，那么C++23之前的标准并没有给出可靠的办法。

这种情况直到C++23标准才得到解决，C++23标准规定编译器必须接受UTF-8编码的源文件作为输入格式，即从C++23标准开始如果我们的源文件格式是UTF-8，那么编译器不能以文件编码问题来拒绝编译。至于如何识别UTF-8，可以由编译器实现自己定义。实际上，主流的编译器都已经支持UTF-8编码，包括GCC，Clang和MSVC。当然了，编译器的实现原因支持更多编码也是可以的，标准并不干涉，但是至少得支持UTF-8。

除了保障可移植性，使用UTF-8作为强制支持的编码的另外一个原因是确保Unicode相关功能可以被广泛使用。比如编写Unicode编码的字符可以让程序和源文件保持一致：

```
char str[]= "😘";
```

上面这段代码中，😘的编码在源文件和程序中都是`F0 9F A5 B0`。

接下来说明一下C++23标准关于UTF-8文件签名（BOM 字节顺序标记）的规定，首先C++标准按照Unicode的建议没有规定强制执行BOM，其次标准规定如果第一个翻译字符是`U+FEFF`，则将其删除，也就是忽略BOM。C++委员会认为应该给实现者足够的自由来处理BOM与编译器标志相矛盾的情况。另一方面，虽然强制BOM识别违背了 Unicode 建议，但这又是一种普遍的做法，因此C++标准既不强制也不阻止使用BOM，只是单纯的忽略它。

最后说明一下主流编译器对于UTF-8编码的识别情况：

- Clang最简单，仅支持UTF-8 并假设所有文件都是UTF-8（至少目前是这样的），忽略BOM。
- GCC是通过`iconv`支持UTF-8，编译标志`-finput-charset=UTF-8`可将源文件解释为UTF-8。默认编码是从 当前环境推断出的，如果无法推断出编码，则回退为UTF-8，同样忽略BOM。
- MSVC通过使用编译标志`/source-charset:UTF-8`支持UTF-8源文件。当有BOM时，MSVC默认使用UTF-8。一般情况下，MSVC假定源文件是以当前用户代码页（Code Page）编码的。

29. 明确==和!=操作符的生成规则

前面的章节提到过在C++20标准中，声明一个`==`操作符函数，`!=`操作符函数会根据前者自动生成。基于这个规则，在C++20标准中有一些情况是没有明确的，例如下面这份代码：

```
struct S
{
    bool operator==(const S &) {
        return true;
    }
    bool operator!=(const S &) {
        return false;
    }
};

int main(){
    bool b = S{} != S{};
}
```

使用Clang15编译这份代码，编译器会发出二义性警告。原因简单来说是`main`函数中调用的`operator!=`即可以是用户自定义的函数重载，也可以是编译器通过`operator==`生成的函数重载，这一点标准是没有明确的。

不仅如此，如果我们将`main`函数中的`operator!=`替换为`operator==`也会引发警告：

```
struct S
{
    bool operator==(const S &) {
        return true;
    }

    bool operator!=(const S &) {
        return false;
    }
};

int main(){
    bool b = S{} == S{};
```

```
}
```

再次使用Clang15编译这份代码，编译器同样会发出二义性警告。这次发出警告的原因和`operator!=`无关，完全是因为`operator==`函数中对比的操作数类型不同带来的。具体来说，编译器针对`bool operator==(const S &)`生成了类似下面的代码：

```
struct S {
    friend bool equal(S, const S &) { ... }
    friend bool equal(const S &, S) { ... }
};
bool b = equal(S{}, S{});
```

很显然，编译器无法从中选择一个最正确的函数，或者说两个函数都是正确的，编译器无法决策，那就选择任意一个，并给出警告。

为了解决上述问题，C++23标准对`==`和`!=`操作符的生成规则做了进一步的明确：如果想通过`operator==(X, Y)`生成`operator!=(X, Y)`和`operator==(Y, X)`，那么请确保仅编写`operator==`函数，并且函数的返回类型为`bool`类型。反之，则需要编写与`operator==`匹配的`operator!=`。

值得注意的是，该规则的提案也是一份C++20缺陷报告，因此使用较新的编译器，例如Clang16，即使使用C++20标准编译上面的两份代码也是可以消除二义性警告的。

30. 修剪行拼接符后的空格

我们都知道，一行代码后如果跟随行拼接符，那么编译器将视源代码换行后的新行和上一行为同一行来解析。不过很少人知道，如果行拼接符后如果跟随了空格，不同编译器的行为是不同的，例如代码：

```
int main() {
  int i = 1
    // \
    + 42;
  return i;
}
```

请注意`// \`这一行的最后有一个空格。这份代码使用GCC、Clang和MSVC的编译结果是不同的。GCC和Clang编译这段代码会修剪行拼接符之后的空格，如此一来，`+ 42`这一行代码会被归类到上一行，即被注释。在这种情况下，变量`i`的最终值为1。而MSVC在编译这段代码的时候并不会修剪行拼接符之后的空格，也就是说`+ 42`仍然是有效的代码，所以变量`i`的最终结果为43。可见两种编译器的实现导致结果差距是巨大的。

为了消除这种差别带来的问题，C++23标准规定，编译器应该修剪行拼接符后的空格，也就是使用GCC和Clang的编译行为。所以标准的修改只会影响到MSVC，不过根据提案文档的描述，通过对vckpg里库的源代码进行的快速分析，发现行拼接符后紧跟空格的情况并不存在，当然了，这并不意味着对MSVC没有影响。

提案文档中还提到了一个有趣的影响点，就是程序员有时会在注释中绘制图表，例如：

```
int main() {
  int hax = 0;
  //  _   _
  // | | | |
  // | |_| | __  __ __
  // |  _  |/ _` \ \/ /
  // | | | | (_| |> <
  // |_| |_|\__,_/_/\_\
  hax = 1337;
  return hax;
}
```

31. 支持#warning预处理指令

对于经常在Linux环境下使用C++开发程序的朋友来说，`#warning`并不是一个新的预处理指令，因为GCC和Clang早就支持该指令了。所以，这一节主要是针对在Windows上使用MSVC开发程序的朋友来介绍这个新引入C++23标准的预处理指令`#warning`。

我们都知道，无论是GCC、Clang还是MSVC，都支持`#error`预处理指令。`#error`预处理指令的作用很简单，停止代码翻译并且从预处理器生成诊断消息，该消息为`#error`该行的标记序列。但是有时候我们并不想阻断代码的编译，只是给出关于代码的一些警告信息。例如我们开发了一个代码库，它可以很好的在64位环境下编译运行，但是在32位环境上有一些性能问题，但是也不会影响程序的正确逻辑，于是我们希望用户能够在32位环境正常编译使用该库并且给出一些警告信息。在这种情况下，我们就需要用到C++23标准引入的`#warning`预处理指令了，该指令与`#error`的功能相比，可以在不停止代码翻译的情况下从预处理器生成诊断消息，诊断信息为`#warning`该行的标记序列。

```
#warning  generate a diagnostic message without stopping translation
#error generate a diagnostic message and stop the translation
```

顺便提一句，如果在C++23标准之前，在MSVC上有类似需求，可以使用`#pragma message("Some message")`，将其完善一下基本上也能完成`#warning`的功能：

```
#define STRINGIZE_HELPER(x) #x
#define STRINGIZE(x) STRINGIZE_HELPER(x)
#define WARNING(desc) message(__FILE__ "(" STRINGIZE(__LINE__) ") : Warning: "
#desc)

#pragma WARNING("Some message")
```

32. 更简单的隐式移动

C++23标准对隐式移动的规则做了简化，一方面针对重载删除了两阶段重载决议，另一方面简化了隐式移动的判定。接下来，我们来具体看看修改了哪些内容。

首先隐式移动的规则变化并不大，只是删除了两阶段重载决策，具体标准描述为：

> 如果表达式符合移动条件，则它是将亡值`xvalue`（见下文）；如果实体是函数、变量、结构化绑定、数据成员或模板参数对象，则为左值；否则为右值；如果标识符指定的是位域它就是位域。
>
> 隐式可移动实体是具有自动存储持续时间的变量，它可以是非易失性对象，也可以是对非易失性对象类型的右值引用。在下面的上下文中标识符表达式符合移动条件：

> - 标识符表达式（可能带括号）是`return`或`co_return`语句的操作数，并且它作为一个可隐式移动的实体被命名在最内层的函数或lambda表达式主体或参数中。
> - 标识符表达式（可能带括号）是`throw`表达式的操作数，并且它作为一个可隐式移动的实体被命名在一个作用域，而该作用域不包含最内层lambda表达式、try-block或function-try-block（如果有）的复合语句或者构造函数初始化器。

删除的两阶段重载规则为：

> 在进行选择拷贝构造函数或return_value重载的重载决议，首先好像表达式或操作数是右值一样。如果第一次的重载决议失败或未执行，那么将再次进行重载决议，这次将表达式或操作数视为左值。

只看规则文字描述不太好理解，我们接下来看一个两阶段重载例子：

```
struct Button {};
struct Editor {};
struct InputBox : Button, Editor {};
struct Window {
    Window(Button&&);
    Window(Editor&&);
    Window(InputBox&);
};

Window getWindow()
{
    InputBox w;
    return w;
}
```

根据标准规则描述，`w`应该先作为右值进行第一次重载决议，它会找到`Window(Button&&)`和`Window(Editor&&)`，很明显这里存在二义性问题，导致第一次重载失败。接下来，将`w`作为左值进行第二次重载决议，找到`Window(InputBox&)`，这里编译器指定C++20标准，可以编译成功。

我们注意到，这种对一个操作数进行两次单独解析的地方在C++标准中只有这样一处，如此特殊又奇怪的规则对于语言来说并不是一件好事，所以C++23标准才会删除这条两阶段重载决议的规则。使用C++23标准，上面的代码会编译报错：

```
error: conversion from 'InputBox' to 'Window' is ambiguous
```

因为`w`是`return`语句的操作数，所以符合移动条件，它是一个将亡值。重载决议时发现`Window(Button&&)`和`Window(Editor&&)`两个重载，于是编译器报告二义性错误。

除了两阶段重载决议的问题，隐式移动在C++20标准还有一处问题在C++23标准中进行了修复。在C++20标准中，隐式可移动实体只会在函数返回对象的时候被判定，如果返回的不是对象，则会出现问题，例如：

```
struct Widget {
    Widget(Widget&&);
};

struct RRefTaker {
    RRefTaker(Widget&&);
};
```

```
RRefTaker foo(Widget&& w) {
    return w;  // 编译成功，返回对象，w触发隐式移动
}

Widget&& bar(Widget&& w) {
    return w;  // 编译失败，非返回对象
}
```

`return w`在函数的返回类型为`RRefTaker`时，隐式可移动实体`w`被视为右值，但当函数的返回类型为`Widget&&`时，`w`被视为左值。

同样的问题也发生在模板推导的时候，例如下面的代码：

```
#include <utility>
struct Widget {};

template<class T>
T&& foo(T&& x) { return x; }

void bar(Widget w) {
    Widget& r = foo(w);                // 编译成功
    Widget&& rr = foo(std::move(w));  // 编译失败
}
```

在这份代码中，因为`w`是一个左值，`Widget& r = foo(w);`这句代码中`T`被推导为`Widget&`，所以返回类型和`x`都是`Widget&`，可以编译成功。但是`Widget&& rr = foo(std::move(w));`中`T`被推导为`Widget`，所以返回类型和`x`都是`Widget&&`，但是由于函数返回类型是`Widget&&`，`x`被视为左值，导致编译失败。

上述问题还会发生在使用`decltype(auto)`的时候：

```
struct Widget {};

Widget val();
Widget& lref();
Widget&& rref();

decltype(auto) foo() {
    decltype(auto) x = val();   // x是Widget类型
    return x;                   // 编译成功，编译器实施拷贝消除
}

decltype(auto) bar() {
    decltype(auto) x = lref();  // x是Widget&类型
    return x;                   // 编译成功，返回类型是Widget&
}

decltype(auto) baz() {
    decltype(auto) x = rref();  // x是Widget&&类型
    return x;                   // 编译失败，返回类型是Widget&&
}
```

关于隐式移动和`decltype(auto)`，在C++20和C++23标准还有一点区别需要重点提出来，请看下面的代码：

```
struct Widget {};

decltype(auto) foo(Widget&& x) {
    return (x);
}

void bar(Widget&& x){}
void bar(Widget& x){}

int main()
{
    bar(foo(Widget()));
}
```

这里的重点是`foo`函数，根据C++20标准`decltype(auto)`的推导规则，这里返回类型被推导为`Widget&`类型。没有发生隐式移动，因为返回类型不是对象类型隐式移动不适用。左值表达式`(x)`愉快地绑定到函数返回类型`Widget&`，调用`void bar(Widget& x)`并且代码编译正常。

不同的是，根据C++23标准，`x`符合移动条件，它是一个将亡值，返回类型被推导为`Widget&&`，将亡值表达式`(x)`愉快地绑定到函数返回类型`Widget&&`，调用`void bar(Widget&& x)`并且代码编译正常。注意，这里返回的是`Widget&&`，而不是C++20标准的`Widget&`。

注意这个修改使得`decltype`在处理括号问题时出现了很多变化：

```
auto f1(int x) -> decltype(x) { return (x); }     // int
auto f2(int x) -> decltype((x)) { return (x); }   // C++20: int&; C++23: 编译失败
auto f3(int x) -> decltype(auto) { return (x); }  // C++20: int&; C++23: int&&
auto g1(int x) -> decltype(x) { return x; }       // int
auto g2(int x) -> decltype((x)) { return x; }     // C++20: int&; C++23: 编译失败
auto g3(int x) -> decltype(auto) { return x; }    // int
```

这里的重点是`f2`、`f3`和`g2`三个函数，其中`f3`使用C++20和C++23标准都可以编译成功，只是返回类型不同，C++20返回类型为`int&`，C++23返回类型为`int&&`。而`f2`和`g2`函数在C++20可以编译成功，返回类型为`int&`，C++23会编译失败，因为返回类型被推导为`int&`，而非常量左值引用无法绑定到右值。

最后，由于隐式移动规则的修改，下面的这份代码以前编译器是给出警告，从C++23标准开始是直接编译失败：

```
int& foo()
{
    int x = 0;
    return x;
}
```

这份代码显然是不正确的，但是过去并没有违反语法规则，因为`x`判定为左值，`foo`的返回类型为左值引用。为了让程序员发现这类代码的问题，贴心的编译器只能给出警告。从C++23标准开始，因为`x`会被判定为右值，非常量左值引用无法绑定到右值，于是编译器报告失败，非常的合理。

33. 静态函数调用运算符函数

C++有一条非常著名的设计原则——零开销原则：

- 我们不应该为不使用的东西付出代价；
- 我们所使用的语言和编译器应该和合理的手工编写的代码一样高效。

简单来说，零开销原则要求C++语言不应该引入任何额外的开销或性能损失，除非明确需要。也意味着C++应该不断的尽可能地高效和精简，以避免不必要的资源消耗。

基于上述这个原则，C++委员会的专家在C++23标准中提出了静态函数调用运算符`static operator()`。

对于有一定C++编程经验的朋友可以回顾一下，是否在经历的项目代码中经常会看到对函数对象（或者叫做仿函数）的定义和使用。对于一个函数对象来说，它的必备条件就是编写一个函数调用运算符`operator()`。很多时候我们会发现，这些函数调用运算符可能根本没有使用对象的数据成员，也就是说该函数和对象本身无关，甚至对象本身就不存在数据成员，是一个无状态的函数对象。

如果是这两种情况，那么函数调用运算符是否传递`this`参数就不再重要了，`operator()`作为非静态成员函数传递`this`参数反而成了一种无用的负担，也就是零开销原则所排斥的。为了明确传递`this`指针带来的消耗，我们来看看下面的例子：

```
#include <vector>
#include <algorithm>
struct X {
    bool operator()(int) const;
    static bool f(int);
};

inline constexpr X x;
```

接下来分为两种方式调用：调用非静态成员函数

```
int count_x(std::vector<int> const& xs)
{
    return std::count_if(xs.begin(), xs.end(), x);
}
```

和调用静态成员函数

```
int count_x(std::vector<int> const& xs)
{
    return std::count_if(xs.begin(), xs.end(), X::f);
}
```

我们使用GCC13和O3优化选项编译上述两份代码会获得这样两份汇编代码：

```
count_x(std::vector<int, std::allocator<int> > const&):
        ...
        mov     esi, DWORD PTR [rbx]
        lea     rdi, [rsp+15]  ; <= 传递this指针
        call    X::operator()(int) const
        cmp     al, 1
        sbb     rbp, -1
        add     rbx, 4
        cmp     r12, rbx
        jne     .L4
```

```
        add     rsp, 16
        mov     eax, ebp
        pop     rbx
        pop     rbp
        pop     r12
        ret
        ...
```

以及

```
count_x(std::vector<int, std::allocator<int> > const&):
        ...
        mov     edi, DWORD PTR [rbx]
        call    X::f(int)
        cmp     al, 1
        sbb     rbp, -1
        add     rbx, 4
        cmp     r12, rbx
        jne     .L4
        mov     eax, ebp
        pop     rbx
        pop     rbp
        pop     r12
        ret
        ...
```

作为非静态成员函数，不可避免的进行了`this`指针的传递。

为了解决上述问题，我们可以使用C++23标准，然后给函数调用运算符添加`static`关键字：

```
struct X {
    static bool operator()(int);
};

inline constexpr X x;

int count_x(std::vector<int> const& xs)
{
    return std::count_if(xs.begin(), xs.end(), x);
}
```

编译的汇编代码就与调用`X::f`一致了：

```
count_x(std::vector<int, std::allocator<int> > const&):
        ...
        mov     edi, DWORD PTR [rbx]
        call    X::operator()(int)
        cmp     al, 1
        sbb     rbp, -1
        add     rbx, 4
        cmp     r12, rbx
        jne     .L4
        mov     eax, ebp
        pop     rbx
```

```
        pop     rbp
        pop     r12
        ret
        ...
```

可能有朋友会提出疑问，针对如此微小的提升，是否值得加入这个特性呢？我们需要从两个方面来解答这个问题，首先，该特性是对原有特性的扩展，并没有改变代码的兼容性，不会带来副作用；其次，虽然单个函数调用看起来用处并不大，但是无论是标准库还是其他项目代码目前都大量的使用了函数对象，函数对象在代码中出现的频率不低，另外函数所处的上下文环境也可能是循环。基于这两点，我认为提升是值得的，并且我们应该鼓励C++不断追求零开销原则的实践。

C++23标准对函数调用运算符的修改不仅仅只停留在用户手动编写的函数调用运算符，对于lambda表达式也同样有效。标准规定，包含`static`的lambda表达式不允许有任何捕获，并且不能同时包含`mutable`和`static`。例如：

```
auto three = []() static { return 3; };
```

这里的重点是，静态lambda表达式必须没有任何捕获，也就是说是一个无状态的lambda表达式。其实针对这个规定是有一定争议的，因为即使捕获了某个对象，但只要在函数体内没有引用该对象就没有传递`this`指针的必要，例如下面的代码：

```
auto under_lock = [lock=std::unique_lock(mtx)]() static {};
```

如果这份代码是合法的，那么它的意义就在于可以在lambda表达式的生命周期内加锁。不过遗憾的是，C++23标准并没有允许这样的用法，理由很简单，不利于教学。当然，如果未来事实证明这个限制是不利于代码编写的，那么将来可能会放宽这个限制，但在C++23标准中，上面的代码是非法的。

值得一提的是，在讨论给lambda表达式增加`static`关键字的会议上，C++委员会的专家曾提出过让编译器的实现自定义无状态lambda表达式的函数调用运算符函数是否为静态函数，例如在编译选项中加入一个标志`-lambda-static`，有了这个编译标志，编译器默认将无状态lambda表达式的函数调用运算符函数编译成静态函数。不过C++委员会很快否定了这个想法，虽然这样做的好处是很容易将以往的无状态lambda表达式的函数调用运算符函数改写为静态函数，但是问题在于程序员并不容易观察到设置编译标志这样一个行为，并且交给实现自定义可能会造成移植问题，使用`static`更容易被程序员察觉并且能够很好的统一不同编译器的行为。