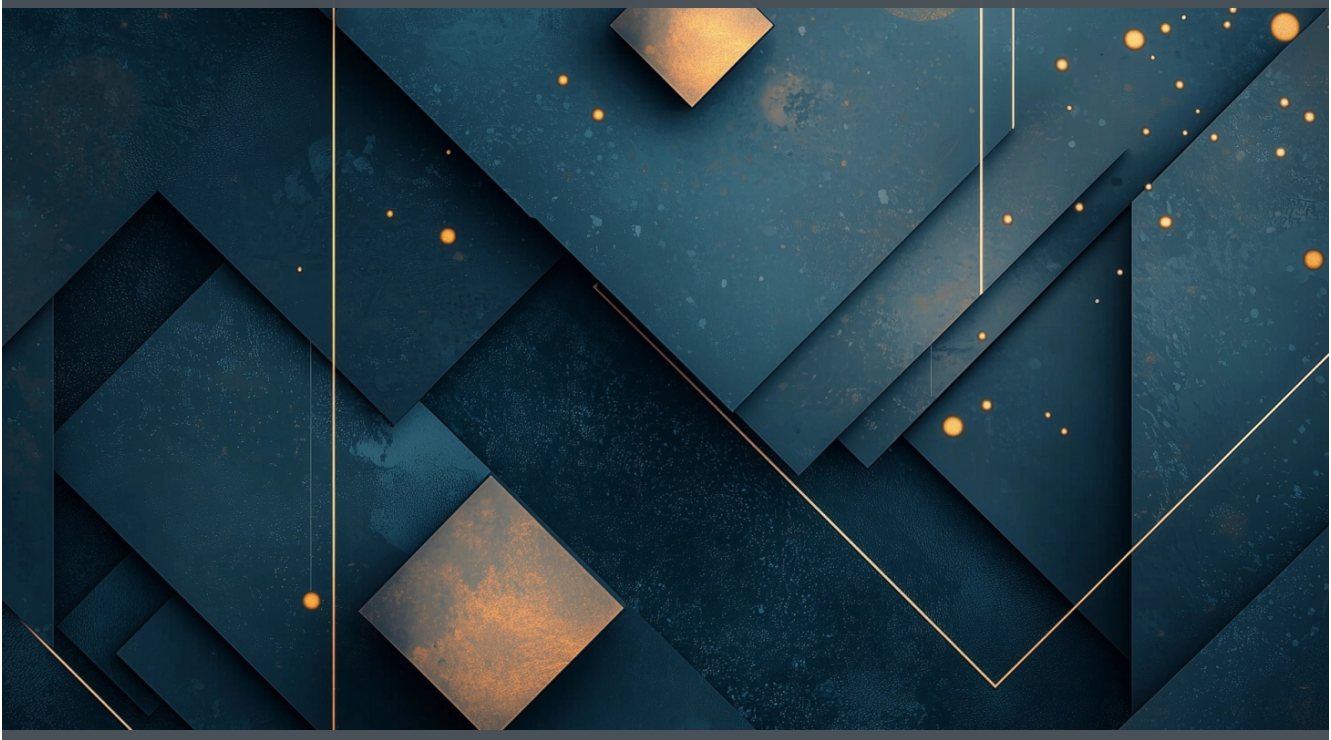
C++ Generative Metaprogramming

Advanced Techniques in Compile-Time Code Generation from C++98 to C++26



CppMore

里缪 著

C++ Generative Metaprogramming

Advanced Techniques in Compile-Time Code Generation from C++98 to C++26

里缪 著

C++ Generative Metaprogramming

版本：第一版
发布日期：2025 年 1 月 5 日
发布地址:`https://github.com/lkimuk/cpp-generative-metaprogramming`
邮箱: lkimuk@outlook.com
封面：Midjourney
错误反馈:任何错误,请反馈至 `https://github.com/lkimuk/cpp-generative-metaprogramming/issues`

C++ Generative Metaprogramming

Version: First Edition
Published: January 5, 2025
Published at `https://github.com/lkimuk/cpp-generative-metaprogramming`
Email: lkimuk@outlook.com
Cover: Midjourney
Error feedback: For any errors, please submit them at `https://github.com/lkimuk/cpp-generative-metaprogramming/issues`

Contents

This page is intentionally left blank.

序

模板奇宏之理，反射注入之技，非初学所能骤窥。其蕴博而奥，其法巧而微，文章虽多，皆止于零碎，体系端末，罕有论者。性情所嗜，心目所寄，数年以来，我于此间颇有见闻。遂取其精要，贯串浃洽，构以体系，裒成此书。分为上中下三篇，每篇数章，曩昔奇技，未来妙法，莫不毕载。

高技丽好，巧艺妙深，字字写来，实为不易。去年秋九月，偶然兴动，我即提笔连载。是年冬，上篇更讫，宏之精微，殊法诡技，尽覆无遗。后诸事相羁，乃搁置不顾，未想就是半载。六月中，中篇始续，窥幽探深，穷理尽妙，模板之术顿显。闲来慢写，历夏经秋，折叠表达式、常量表达式等章继卒，繁篇方毕。渐又霜风凄紧，年关迫近，忙处即思，得暇即写，日就月将，下篇终成矣。正是光阴似水，日月如梭，指尖过处，尽是心酸。

本书所涉之元编程技术，利在写库，非三五年经验不可尽识。诸般秘奥，兼涵并包，然常识概念，平奇技法，尽皆略去，故不适合初学者。至于阅读之要，每行代码，小大靡遗，纤悉洞了，臻乎无惑，必能雾开日莹，尘尽鉴明。

驰光不留，一岁倏忽，思来每年，飘蓬形影，长为远客。世事千变，而志趣不衰，实是难得，因作《满庭芳・行迈》以寄慨，词曰：

花弄春柔，轻晖薰柳，一年无限风光。云萍羁旅，辗转太匆忙。
巷陌人间旧事，饱经惯、烟雨寻常。忽回首，凌云意气，点点落寒窗。

微凉。浑似水，漂流容易，不耐天长。更何须争个，谁弱谁强。
平日愁遥思往，牵绪乱、最是难当。从前望，轻衣便履，缓缓步斜阳。

里缪
2024 年 12 月 22 日

前言

元编程，是对于编程的编程，它能够读取、生成、分析或修改源代码本身，编写抽象层次更高的代码。C++ 中谈论元编程，一般是指处于编译期的编程，其发展可以分成三个阶段：

1. 模板阶段：以模板作为编译期对类型和值进行计算、生成、变换的工具，具有独立的泛型编程体系，采用函数式范式，通过不可变数据来避免状态变化，通过特化和递归来控制程序流程，通过约束模板参数来表达定制点，所有操作必定发生于编译期。由于状态无法发生变化，模板编程只能使用常量，为了表达变化，各类奇技淫巧层出不穷，逐渐诞生了一套独立的编程技术。这些“太过聪明”的技巧，必须对模板元编程具有极其深入的理解，才能编写和维护，门槛很高。

2. 常量表达式阶段：以 `constexpr` 作为编译期计算的工具，由来已久，模板发展稳定后才将重心转来发展此种元编程方式，用来在常量求值阶段产生编译期值。结合现有的编程体系而来，采用命令式范式，通过变量赋值来表达频繁变化的状态，通过条件和循环来控制程序流程，通过函数重载来表达定制点，所有操作并不一定发生于编译期。由于支持变量、条件和循环，这种方式与现有的运行期编程方式并无太大的差异，所以简化了部分模板元编程的复杂度，降低了编译期编程的难度。

3. 静态反射阶段：以静态反射和源码注入作为编译期计算的工具，等待既久，终泛涟漪。模板和常量表达式只是元编程系统的基础，静态反射才是其中的核心，这是一个全新的阶段，一个完善、灵活、强大的元编程时期。一套语言级别的类型操纵和代码生成机制将给元编程带来翻天覆地的变化，能够编写抽象层次更高的代码，也能够设计灵活性更强的程序，更好地封装变化。

产生式元编程，则专门是指具备代码生成能力的元编程。所谓代码生成能力，就是能够以代码来生成代码的能力，以上三个阶段的元编程特性都能够在

编译期生成代码，只是对代码生成的可控制程度不同而已。除了 C++，C 中的宏也拥有代码生成能力，不过只是一种最简陋的的特性，宏元编程的门槛甚至比模板元编程还要高，工具太过简单，结果就是最简单功能也需要从头演绎，充满着殊法诡技，学来不易。

这些不同的产生式元编程工具，构成了本书的所有内容。结构上分为上中下三篇，上篇主讲宏元编程，分布于第一至四章；中篇主讲模板元编程和常量表达式元编程，分布于第五至八章；下篇主讲反射元编程，分布于最后两章。上篇其实是 C 的特性，中篇才是纯 C++ 目前能够使用的特性，是核心中的核心，所以篇幅最长，下篇是 C++ 未来的特性，不知何时才能完全支持书中介绍的内容。

本书目标读者为已经具备多年 Modern C++ 经验的高级开发者，所以不会从零讲解各种次要特性的来龙去脉，一些概念，也是拿来即用，不作赘述，以保证用较短的篇幅和时间来写完想要表达的内容。若是初/中级读者，本书也许还不太适合，读起来会比较吃力，产生式元编程技术在寻常开发项目中基本不会用到，通常用于写库或框架，可先收藏起来，需要之时再来学习。

里缪

2024 年 12 月 22 日

致谢

本书待成之时，等疾风帮忙校对了第一至四章，何荣华校对了第五、六章，邹启翔校对了第七、八章，修复了一些编写错误，使各章内容更加准确。谨致谢意！

写作过程中，亦阅读了不计其数的网络文章、问答和 C++ 提案，在此难以一一列出，对这些作者也一并表示感谢。

也要感谢 2024 年的我，这一年心性和雅，神清志逸，高效完成了多个目标。望来年志趣不改，勇敢探索，创造出更多优秀的作品。

里缪

2025 年 1 月 3 日

This page is intentionally left blank.

Chapter 1

Basics of Macros

作为 C++ 元编程的核心工具，宏在 C 时期便已存在，这种代码生成工具只是基于最简单的代码替换，功能十分有限。即便是像条件判断、循环、递归这些最基本的编程结构，在宏系统中都是专家级的技术，充满着殊法诡巧，人们常谓这种代码为“天书”，足见晦涩程度。

只是，举目可观的三五年内，依旧没有对标的新特性能够完全替代宏这种原始的代码生成工具。各类库中，难以避免使用宏来生成一些具有可复用性的代码，稍大一点儿的项目也免不得需要借助宏来简化一些逻辑。因此，本书便以宏作为开篇，深入讲解一下其中的高级知识。

本章亦是上篇的起始之章，上篇共包含四章内容，难度绝对不低，需要对宏和模板的基础内容都已具备一定的了解。

上篇中，第一章和第二章主要集中于讲 What 和 How，不会涉及核心的语言规则，倘若此前不曾接触过宏的高级技法，这两章突然遭到众多崭新思路和精密技巧的冲击，难免会有挫败感。不要纠结，不要停留，首次阅读，了解是什么和怎么做就已经达到了目的。继续往下浏览，第三章会集中讲 Why，前面不懂的底层原理和语言规则都会详细解释，至此便能拨云见日。

闲言少叙，正式开始宏元编程的内容吧！

1.1 The Problem

宏元编程须从可变宏参数开始谈起，可变模板参数的个数可以通过 `sizeof...(args)` 获取，宏里面如何操作呢？便是本章的问题。

接下来的各节，便逐步分析这个问题，实现需求，其间穿插各种宏元编程的原理。默认使用的编译器为 GCC，宏的基本特性很古老，大多版本都支持。

1.2 Analysis and Implementation

宏只有替换这一个功能，所谓的复杂代码生成功能，都是基于这一核心理念演绎出来的。故小步慢走，循序渐进，便可降低理解难度。

问题是获取宏参数包的个数，第一步应该规范过程。

过程的输入是一系列对象，对象类型和个数是变化的；过程的输出是一个值，值应该等于输入对象的个数。如果将输入替换为任何类型的其他对象，只要个数没变，结果也应该保持不变。

于是，先通过宏函数表示出过程原型：

```
#define COUNT_VARARGS(...) N
```

根据宏的唯一功能可知，输出只能是一个值。依据此点，便可以否定遍历、迭代之类的常规思路。可以考虑多加一层宏，通过由特殊到普遍的思想来不断分析，推出最终结果：

```
#define GET_VARARGS(a) 1
#define COUNT_VARARGS(...) GET_VARARGS(__VA_ARGS__)
```

目前这种实现只能支持一个参数的个数识别，可以通过假设，在特殊的基础上逐次增加参数。于是得到：

```
#define GET_VARARGS(a, b) 2
#define GET_VARARGS(a)    1
#define COUNT_VARARGS(...) GET_VARARGS(__VA_ARGS__)
```

若假设成立，通过暴力法已经能够将特殊推到普遍，问题旋即解决。但是，宏并不支持重载，有没有可能实现呢？通过再封装一层，消除名称重复，得到：

```
#define GET_VARARGS_2(a, b) 2
#define GET_VARARGS_1(a)    1
#define GET_VARARGS(...)    GET_VARARGS_X(__VA_ARGS__)
#define COUNT_VARARGS(...)  GET_VARARGS(__VA_ARGS__)
```

至此可知，若要实现宏重载效果，必须确定 `GET_VARARGS_X` 中的 `X`，而它又与参数个数相同，绕了一圈，问题又回到起点，说明此路不通。

因此，回到特殊情况重新分析，函数名称已确定不能重复，唯一能够改变的就只剩下输入和输出。既然无法重载，那么输出也就不能直接写出，先同时改变输入和输出，满足特殊情况再说：

```
#define GET_VARARGS(N, ...) N
#define COUNT_VARARGS(...)  GET_VARARGS(1, __VA_ARGS__)
```

已经支持一个参数，再尝试写出两个参数的情况：

```
#define GET_VARARGS(N1, N2, ...) N?
#define COUNT_VARARGS(...) GET_VARARGS(1, 2, __VA_ARGS__)
```

输出存在变化，`N?` 却只能是一个唯一确定的值，无法适应这种变化性，这种尝试也以失败告终。于是，排除改变输出的可能性，只余下输入是可以改变的。继续尝试：

```
#define GET_VARARGS(N1, N2, ...) N2
#define COUNT_VARARGS(...) GET_VARARGS(__VA_ARGS__, 1, 2)
```

稍微改变输入顺序，便有了新的发现：当 `__VA_ARGS__` 个数为 1 时，`N2` 为 1；为 0 时，`N2` 为 2。这表明间接层的输入参数之间存在着某种关系，接着扩大样本，寻找规律：

```
#define GET_VARARGS(N1, N2, N3, N4, N5, ...) N5
#define COUNT_VARARGS(...) \
    GET_VARARGS(__VA_ARGS__, 1, 2, 3, 4, 5)
```

列出 Table 1.1 分析不同参数个数的替换结果。

Table 1.1: `GET_VARIABLES`: Arguments and Replacements

输入参数个数	N5
0	5
1	4
2	3
3	2
4	1

由此可知，参数个数和输出顺序相反且少 1。故修改实现为：

```
#define GET_VARARGS(N1, N2, N3, N4, N5, ...) N5
#define COUNT_VARARGS(...) GET_VARARGS(__VA_ARGS__, 4, 3, 2,
↪  1, 0)
```

通过发现的规律，我们实现了由特殊到普遍的过程。函数的参数个数一般是有限的，只要再通过暴力法扩大数值范围，便能够为该需求实现一个通用的工具。

1.3 Testing and Optimization

普遍性的解决方案还需要经历实践的检验，暴露实现当中可能存在的技术问题。当前实现中的 `__VA_ARGS__` 就是一个问题，当输入参数的个数为 0 时，替换之后会留下一个“,”，这又打破了普遍性。

通过查阅资料，发现 `##__VA_ARGS__` 可以消除这种情况下的逗号，但是不能写在参数的开头。根据这个约束，改变参数，进一步优化实现。得到：

```
#define GET_VARARGS(Ignored, N1, N2, N3, N4, N5, ...) N5
#define COUNT_VARARGS(...) \
    GET_VARARGS("ignored", ##__VA_ARGS__, 4, 3, 2, 1, 0)
```

至此，该实现终于具备普遍性。接着，整理接口名称，使其更加规范。变为：

```
#define GET_VARARGS(_0, _1, _2, _3, _4, N, ...) N
#define COUNT_VARARGS(...) \
    GET_VARARGS("ignored", ##__VA_ARGS__, 4, 3, 2, 1, 0)
```

在此基础上，将它由 4 推到 20、50、100……都不成问题，只要选择一个合适够用的数值就行。

再进一步检测，将其扩展到其他编译器进行编译，并尝试切换不同的语言版本编译，观察结果是否一致。经过检测，发现 `##__VA_ARGS__` 的行为并不通用，不同语言版本和编译器的结果都不尽相同。此时，就需要进一步查找解决方案，增强实现的通用性。

最终查找到 C++20 的 `__VA_OPT__` 已经解决了这个问题，遂替换实现：

```
#define GET_VARARGS(_0, _1, _2, _3, _4, N, ...) N
#define COUNT_VARARGS(...) GET_VARARGS( \
    "ignored" __VA_OPT__(,) __VA_ARGS__, 4, 3, 2, 1, 0)

int main() {
    printf("zero arg: %d\n", COUNT_VARARGS());
    printf("one arg: %d\n", COUNT_VARARGS(1));
    printf("two args: %d\n", COUNT_VARARGS(1, 2));
    printf("three args: %d\n", COUNT_VARARGS(1, 2, 3));
    printf("four args: %d\n", COUNT_VARARGS(1, 2, 3, 4));
}
```

若要支持 C++20 之前的版本，那么只需要再寻找办法，增加预处理即可。

1.4 Enhancing Generality

经上述分析实现，“计算可变宏参数个数” 的需求已基本由 COUNT_VARARGS 实现。在此，先总结一下用到的思想和技术，再继续往下走：

[1.1] 通过增加一个间接层，能够解决无法直接解决的问题。

[1.2] 小步快走，由特殊逐渐扩展到普遍，能够降低问题的解决难度。

[1.3] 规范过程，确认变与不变，逐步控制变量，能够全面分析问题。

[1.4] 尝试改变输入的顺序、大小、个数……也许会有新的发现。

[1.5] 初步发现规律时，扩大样本验证，能够将特殊推广到普遍。

[1.6] 扩展编译器及语言版本，能够更全面地测试解决方案的普遍性。

[1.7] 可变宏参数作为输入和欲输出结果组合起来，其间规律可以表达条件逻辑。

掌握了这些解决问题的思路，下面就来进一步完善解决方案，使 COUNT_VARARGS 支持 C++20 以下版本。

问题的关键在于 ##__VA_ARGS__ 不具备通用性，此时一种精妙的技术是分情况讨论，即零参和多参分别处理。考虑起初分析时的一条失败道路：

```
#define GET_VARARGS_2(a, b) 2
#define GET_VARARGS_1(a)    1
#define GET_VARARGS(...)    GET_VARARGS_X(__VA_ARGS__)
#define COUNT_VARARGS(...)  GET_VARARGS(__VA_ARGS__)
```

这是函数重载的思路，因必须确定 GET_VARARGS_X 中的 X，而 X 又和可变宏参数相关，可变宏参数又属于无限集，遂无法确定这个 X，致此路不通。但若是改变前提条件，使 X 的值属于有限集，这条路便可以走通，这里便能够利用起来。只考虑零参和多参的情况，X 的取值范围就可以限定在 0 和 1 之间。只需设法判断可变宏参数属于哪种情况，便可借此实现重载，分发处理逻辑。

根据假设，写出基本代码：

```
#define _COUNT_VARARGS_0(...) N
#define _COUNT_VARARGS_1()    0
#define COUNT_VARARGS_0(...) _COUNT_VARARGS_0(__VA_ARGS__)
#define COUNT_VARARGS_1(...) _COUNT_VARARGS_1()
#define OVERLOAD_INVOKE(call, version) call ## version
#define COUNT_VARARGS(...) OVERLOAD_INVOKE( \
    COUNT_VARARGS_, IS_EMPTY(__VA_ARGS__))
```

定义两个重载宏函数 `COUNT_VARARGS_1` 和 `COUNT_VARARGS_0`，前者处理无参情况，后者处理多参情况。如此一来，空包时 `IS_EMPTY` 返回 1，调用分发到 `COUNT_VARARGS_1`，最终结果直接置为 0；多参时 `IS_EMPTY` 返回 0，调用分发到 `COUNT_VARARGS_0`，使用前面的方法处理即可。

于是，主要问题就转换为如何实现 `IS_EMPTY`。根据结论 [1.3] 分析新的需求，它的输入是可变宏参数，过程是判断空或非空，结果是 1 或 0。过程依旧是条件逻辑，而结论 [1.7] 正好可以表达条件逻辑，初步实现思路这便有了。

根据设想，写出基础代码：

```
#define HAS_COMMA(_0, _1, _2, ...) _2
#define IS_EMPTY(...) HAS_COMMA(__VA_ARGS__, 0, 1)
```

空包时，第一步替换结果为 `HAS_COMMA(, 0, 1)`，第二步替换结果为 1。列出 Table 1.2，分析更多样本。

Table 1.2: Input Analysis of `IS_EMPTY()`

IS_EMPTY() 输入	替换结果
	1
1	1
,	0
1,2	0

可以发现，空包和 1 个参数情况的结果相等，为 1；多参情况的结果相等，为 0。扩大 `HAS_COMMA` 的参数之后，结果依然成立：

```
#define HAS_COMMA(_0, _1, _2, _3, _4, _5, ...) _5
#define IS_EMPTY(...) HAS_COMMA(__VA_ARGS__, 0, 0, 0, 0, 1)
```

如今难题变成如何区分空包和 1 个参数，这个问题依旧不好处理。宏只具备替换能力，可以尝试将空包替换成多参的同时，保持 1 个参数不变，这样就能够达到区分的目的。下面的代码用于说明这个技巧：

```
#define _TRIGGER_PARENTHESIS(...) ,
#define TEST(...) _TRIGGER_PARENTHESIS __VA_ARGS__ ()
```

若参数为空，`TEST()` 第一步被替换为 `_TRIGGER_PARENTHESIS ()`，第二步被替换为 `,`。而 `,` 等价于 `a, b`，属于两个参数，便达到了将空包变为多参的需求。若是参数为 1，`TEST` 第一步被替换为 `_TRIGGER_PARENTHESIS 1 ()`，宏不会继续展开，只要不使用该参数，它也不会报错，还是当作 1 个参数。如此一来，问题迎刃而解，开始将基础代码合并:

```
#define _TRIGGER_PARENTHESIS(...) ,
#define _COMMA_CHECK(_0, _1, _2, _3, _4, _5, ...) _5
#define HAS_COMMA(...) \
    _COMMA_CHECK(__VA_ARGS__, 0, 0, 0, 0, 1)
#define IS_EMPTY(...) \
    HAS_COMMA( _TRIGGER_PARENTHESIS __VA_ARGS__ () )
```

重新列出 Table 1.3 分析（非最终分析版）。

Table 1.3: Input Analysis of `IS_EMPTY()`

IS_EMPTY() 输入	替换结果
	0
1	1
,	0
1,2	0
()	0

区分空包和 1 个参数的问题完美解决！暂停下来，调整接口，让它更加规范。一般 1 为真，0 为假，而现在 1 和 0 的意义完全相反，难以顾名思义，先把它改变过来。调整代码为：

```
#define _TRIGGER_PARENTHESIS(...) ,
#define _COMMA_CHECK(_0, _1, _2, _3, _4, _5, ...) _5
#define HAS_COMMA(...) \
    _COMMA_CHECK(__VA_ARGS__, 1, 1, 1, 1, 0)
#define IS_EMPTY(...) \
    HAS_COMMA( _TRIGGER_PARENTHESIS __VA_ARGS__ () )
```

调整后的表格为 Table 1.4。

随后分析新产生的问题，可以发现多参之间又无法区分是否为空包。解决这个问题的思路是多条件约束，从结果集中依次剔除不满足的元素。先把使用技巧前后的表格汇总起来，寻找规律，如 Table 1.5。

Table 1.4: Input Analysis of Optimized `IS_EMPTY()`

IS_EMPTY() 输入	替换结果
	1
1	0
,	1
1,2	1
()	1

Table 1.5: Analyzing `HAS_COMMA()` Under Two Conditions

HAS_COMMA() 参数/输入		1	,	1,2	()
__VA_ARGS__	0	0	1	1	0
_TRIGGER_PARENTHESIS __VA_ARGS__ ()	1	0	1	1	1

这两个条件组合起来，便可进一步排除包含 , 的情况。在此基础上，充分利用排列组合，再增加了两个条件，表格变为 Table 1.6。

Table 1.6: Analyzing `HAS_COMMA()` Under Four Conditions

HAS_COMMA() 参数/输入		1	,	1,2	()
__VA_ARGS__	0	0	1	1	0
_TRIGGER_PARENTHESIS __VA_ARGS__	0	0	1	1	1
__VA_ARGS__ ()	0	0	1	1	0
_TRIGGER_PARENTHESIS __VA_ARGS__ ()	1	0	1	1	1

首先，使用 `_TRIGGER_PARENTHESIS __VA_ARGS__ ()` 区分了空包和 1 参的情况，得到的结果集包含空包和多参。其次，借助 `__VA_ARGS__` 区分了空包和 `,`（即为多参）的情况，两相组合，结果集便只剩下空包。但还有一些特殊情况，比如第一个输入参数包含 `()`，此时 `HAS_COMMA((1))` 展开为 `HAS_COMMA(, ())`，一个参数替换后变成两个参数会造成误判，以 `_TRIGGER_PARENTHESIS __VA_ARGS__` 能够进一步排除这类结果。最后，如果输入为一个宏或无参函数，例如 `HAS_C」OMMA(Foo)`，替换后就变成了 `HAS_COMMA(_TRIGGER_PARENTHESIS Foo ())`，结果可能会出乎意料，通过 `__VA_ARGS__ ()` 能够排除这类结果。

对于无参宏/函数，如 `#define Foo() 1, 2`，替换结果如 Table 1.7 所示。

这四个条件组合起来，也就是说四个条件的结果为 **0001**，就可以唯一确认

Table 1.7: Input Analysis of `HAS_COMMA()`

HAS_COMMA() 参数/输入		1	,	1,2	()	Foo
__VA_ARGS__	0	0	1	1	0	0
_TRIGGER_PARENTHESIS __VA_ARGS__	0	0	1	1	1	0
__VA_ARGS__ ()	0	0	1	1	0	1
_TRIGGER_PARENTHESIS __VA_ARGS__ ()	1	0	1	1	1	1

空包。现在便可运用老办法，添加重载，当重载为 `0001` 时，处理空包，返回 1，其他所有情况返回 0。实现如下：

```
#define _COMMA_CHECK(_0, _1, _2, _3, _4, _5, \
    _6, _7, _8, _9, _10, R, ...) R
#define HAS_COMMA(...) _COMMA_CHECK(__VA_ARGS__, \
    1, 1, 1, 1, 1, 1, 1, 1, 1, 0)

#define _IS_EMPTY_CASE_0001 ,
#define _IS_EMPTY_CASE(_0, _1, _2, _3) \
    _IS_EMPTY_CASE_ ## _0 ## _1 ## _2 ## _3
#define _IS_EMPTY_IMPL(_0, _1, _2, _3) \
    HAS_COMMA(_IS_EMPTY_CASE(_0, _1, _2, _3))
#define IS_EMPTY(...) \
    _IS_EMPTY_IMPL( \
        HAS_COMMA(__VA_ARGS__), \
        HAS_COMMA(_TRIGGER_PARENTHESIS_ __VA_ARGS__), \
        HAS_COMMA(__VA_ARGS__ ()), \
        HAS_COMMA(_TRIGGER_PARENTHESIS_ __VA_ARGS__ ()) \
    )
```

四个条件的处理结果进一步传递到 `_IS_EMPTY_IMPL`，它又通过 `_IS_EMPTY_CASE` 组装成重载特化 `_IS_EMPTY_CASE_0001`。其他所有情况都没有定义相应的重载，因而经由 `HAS_COMMA` 调用 `_COMMA_CHECK` 的参数个数都相同，最终只会返回 0。而 `_IS_EMPTY_CASE_0001` 被替换为 `,`，相当于参数个数加一，最终返回 1。

子问题解决，现在回到原问题，看看最初的实现：

```
#define _COUNT_VARARGS_0(...) N
#define _COUNT_VARARGS_1() 0
#define COUNT_VARARGS_0(...) _COUNT_VARARGS_0(__VA_ARGS__)
#define COUNT_VARARGS_1(...) _COUNT_VARARGS_1()
#define OVERLOAD_INVOKE(call, version) call ## version
#define COUNT_VARARGS(...) OVERLOAD_INVOKE( \
    COUNT_VARARGS_, IS_EMPTY(__VA_ARGS__))
```

`IS_EMPTY` 只有空包时才会返回 1，其他情况都返回 0。因此当前的重载版本已经可以使用，空包时最终的参数直接由 `_COUNT_VARARGS_1` 将结果替换为 0。对于 `_COUNT_VARARGS_0` 多参版本，沿用上节的实现即可：

```
#define _COUNT_VARARGS_0_IMPL(_0, _1, _2, _3, _4, N, ...) N
#define _COUNT_VARARGS_0(...) \
    _COUNT_VARARGS_0_IMPL(__VA_ARGS__, 5, 4, 3, 2, 1)
```

到这一步，这个问题就解决了。根据总结 [1.6]，进一步切换编译器检验一下方案的普遍性，发现 MSVC 结果错误。原来，MSVC 每次传递 `__VA_ARGS__` 时，都会将其当作整体传递。例如：

```
#define A(x, ...) x and __VA_ARGS__
#define B(...) A(__VA_ARGS__)

B(1, 2); // 替换为 1, 2 and
```

一种解决方案是强制宏展开，实现很简单：

```
#define EXPAND(x) x
#define A(x, ...) x and __VA_ARGS__
#define B(...) EXPAND( A(__VA_ARGS__) )

B(1, 2); // 替换为 1 and 2
```

为了适配 MSVC，我们需要将所有传递的 `__VA_ARGS__` 都强制展开。最终的实现为：

```
#include <cstdio>

#define EXPAND(x) x
#define _TRIGGER_PARENTHESIS(...) ,

#define _COMMA_CHECK(_0, _1, _2, _3, _4, _5, _6, _7, _8, \
    _9, _10, R, ...) R
#define HAS_COMMA(...) EXPAND( _COMMA_CHECK(__VA_ARGS__, \
    1, 1, 1, 1, 1, 1, 1, 1, 1, 0) )

#define _IS_EMPTY_CASE_0001 ,
#define _IS_EMPTY_CASE(_0, _1, _2, _3) \
    _IS_EMPTY_CASE_ ## _0 ## _1 ## _2 ## _3
#define _IS_EMPTY_IMPL(_0, _1, _2, _3) \
```

```
    HAS_COMMA(_IS_EMPTY_CASE(_0, _1, _2, _3))
#define IS_EMPTY(...) \
    _IS_EMPTY_IMPL( \
        EXPAND( HAS_COMMA(__VA_ARGS__) ), \
        EXPAND( HAS_COMMA(_TRIGGER_PARENTHESIS __VA_ARGS__)
↪   ), \
        EXPAND( HAS_COMMA(__VA_ARGS__ ()) ), \
        EXPAND( HAS_COMMA(_TRIGGER_PARENTHESIS __VA_ARGS__
↪   ()) ) \
    )

#define _COUNT_VARARGS_0_IMPL(_0, _1, _2, _3, _4, N, ...) N
#define _COUNT_VARARGS_0(...) EXPAND( \
    _COUNT_VARARGS_0_IMPL(__VA_ARGS__, 5, 4, 3, 2, 1) )
#define _COUNT_VARARGS_1() 0
#define COUNT_VARARGS_0(...) EXPAND( \
    _COUNT_VARARGS_0(__VA_ARGS__) )
#define COUNT_VARARGS_1(...) _COUNT_VARARGS_1()
#define OVERLOAD_INVOKE(call, version) call ## version
#define OVERLOAD_HELPER(call, version) \
    OVERLOAD_INVOKE(call, version)
#define COUNT_VARARGS(...) EXPAND( \
    OVERLOAD_HELPER(COUNT_VARARGS_, \
    IS_EMPTY(__VA_ARGS__))(__VA_ARGS__) )

int main() {
    // 0 1 2 3 4
    printf("%d %d %d %d %d\n",
        COUNT_VARARGS(),
        COUNT_VARARGS(1),
        COUNT_VARARGS('a', 'b'),
        COUNT_VARARGS('a', 'b', 'c'),
        COUNT_VARARGS('a', 'b', 1, 2));
}
```

这种解决方案不仅支持 C++ 所有版本，而且在 C 中也能使用。

1.5 Merging Solutions

最后，将两种方案合并起来，便能得到一个通用的实现。借助预处理分语言版本选择不同的实现，C++20 以上使用新的简洁做法，C++20 以下使用这种通用但复杂的做法。

```
#include <cstdio>

#if defined(__cplusplus) && __cplusplus >= 202002L
    #define GET_VARARGS(_0, _1, _2, _3, _4, N, ...) N
    #define COUNT_VARARGS(...) GET_VARARGS( \
        "ignored" __VA_OPT__(,) __VA_ARGS__, 4, 3, 2, 1, 0)
#else
    #define EXPAND(x) x
    #define _TRIGGER_PARENTHESIS(...) ,

    #define _COMMA_CHECK(_0, _1, _2, _3, _4, _5, _6, _7, \
        _8, _9, _10, R, ...) R
    #define HAS_COMMA(...) EXPAND( _COMMA_CHECK( \
        __VA_ARGS__, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0) )

    #define _IS_EMPTY_CASE_0001 ,
    #define _IS_EMPTY_CASE(_0, _1, _2, _3) \
        _IS_EMPTY_CASE_ ## _0 ## _1 ## _2 ## _3
    #define _IS_EMPTY_IMPL(_0, _1, _2, _3) \
        HAS_COMMA(_IS_EMPTY_CASE(_0, _1, _2, _3))
    #define IS_EMPTY(...) \
        _IS_EMPTY_IMPL( \
            EXPAND( HAS_COMMA(__VA_ARGS__) ), \
            EXPAND( HAS_COMMA(_TRIGGER_PARENTHESIS
↪   __VA_ARGS__) ), \
            EXPAND( HAS_COMMA(__VA_ARGS__ ()) ), \
            EXPAND( HAS_COMMA(_TRIGGER_PARENTHESIS
↪   __VA_ARGS__ ()) ) \
```

```
        )

    #define _COUNT_VARARGS_0_IMPL( \
        _0, _1, _2, _3, _4, N, ...) N
    #define _COUNT_VARARGS_0(...) EXPAND( \
        _COUNT_VARARGS_0_IMPL(__VA_ARGS__, 5, 4, 3, 2, 1) )
    #define _COUNT_VARARGS_1() 0
    #define COUNT_VARARGS_0(...) EXPAND( \
        _COUNT_VARARGS_0(__VA_ARGS__) )
    #define COUNT_VARARGS_1(...) _COUNT_VARARGS_1()
    #define OVERLOAD_INVOKE(call, version) call ## version
    #define OVERLOAD_HELPER(call, version) \
        OVERLOAD_INVOKE(call, version)
    #define COUNT_VARARGS(...) EXPAND( \
        OVERLOAD_HELPER(COUNT_VARARGS_, \
        IS_EMPTY(__VA_ARGS__))(__VA_ARGS__) )
#endif

int main() {
    printf("version: %ld\n", __cplusplus);

    // 0 1 2 3 4
    printf("%d %d %d %d %d\n",
        COUNT_VARARGS(),
        COUNT_VARARGS(1),
        COUNT_VARARGS('a', 'b'),
        COUNT_VARARGS('a', 'b', 'c'),
        COUNT_VARARGS('a', 'b', 1, 2));
}
```

1.6 Summary

本章以可变宏参数计数为例，穿插讲解宏编程的核心原理，又涵盖问题分析思路与完善步骤，小步慢走，终是实现需求。

掌握了这些核心技术，才能利用宏来表示基本逻辑，才能演绎出诸多新工

具，反过来简化如今的实现。

其间细节，更深层次的原理，以及更多技术，将在后续的两个章节呈现。

Chapter 2

Conditionalzs, Loops, and Recursion in Macros

历观编程概念，避及重复，提其不变，于迭代与递归为甚。产生式元编程，乃欲自动产生百千代码，迭代递归，自是重要组件，不可不备。

本章以 FOR_EACH 为例，渐次派生问题，引出技术要点，实现强大组件。

2.1 Iteration

FOR_EACH 用来迭代数据集合，依次取出数据，后调用函数处理。参数易定，一个回调函数加上可变参数。代码表示：

```
#define FOR_EACH(call, ...)

// Invoke
FOR_EACH(Foo, 1, 2, 3)
```

实现需逐个取出可变宏参数，调用 Foo() 处理。第一章已经讲解过宏重载的实现方法，这也是条件逻辑在宏中的表示法。无循环时，递归思想便是拆解参数包的唯一方式，将重载实现法与递归思想结合起来，即可达逐个取参数之效。

遂可如此表示：

```
#define _FOR_EACH_1(call, x) call(x)
#define _FOR_EACH_2(call, x, ...) \
```

```
    call(x) _FOR_EACH_1(call, __VA_ARGS__)
#define _FOR_EACH_3(call, x, ...) \
    call(x) _FOR_EACH_2(call, __VA_ARGS__)
#define _FOR_EACH_4(call, x, ...) \
    call(x) _FOR_EACH_3(call, __VA_ARGS__)
#define _FOR_EACH_5(call, x, ...) \
    call(x) _FOR_EACH_4(call, __VA_ARGS__)

#define FOR_EACH(call, ...) \
    OVERLOAD_INVOKE(_FOR_EACH, \
    COUNT_VARARGS(__VA_ARGS__))(call, __VA_ARGS__)
```

`OVERLOAD_INVOKE()` 在上章的基础上已部分优化，如今的实现变为：

```
#define CONCAT_HELPER(a, b) a ## b
#define CONCAT(a, b) CONCAT_HELPER(a, b)

#define OVERLOAD_INVOKE_1(call, v1) \
    CONCAT(CONCAT(call, _), v1)
#define OVERLOAD_INVOKE_2(call, v1, v2) \
    CONCAT( CONCAT(OVERLOAD_INVOKE_1(call, v1), _), v2 )
#define OVERLOAD_INVOKE_3(call, v1, v2, v3) \
    CONCAT( CONCAT(OVERLOAD_INVOKE_2(call, v1, v2), _), v3)
#define OVERLOAD_INVOKE_4(call, v1, v2, v3, v4) \
    CONCAT( CONCAT(OVERLOAD_INVOKE_3(call, v1, v2, v3), _),
↪   v4)
#define OVERLOAD_INVOKE_5(call, v1, v2, v3, v4, v5) \
    CONCAT( CONCAT(OVERLOAD_INVOKE_4(call, v1, v2, v3, v4),
↪   _), v5)
#define OVERLOAD_INVOKE(call, ...) \
    CONCAT( OVERLOAD_INVOKE_, \
    COUNT_VARARGS(__VA_ARGS__) )(call, __VA_ARGS__)
```

因其使用太过广泛，太过频繁，于是提前重新封装成更适用的函数。它基于 `COUNT_VARARGS()` 实现，而在第一章中，`COUNT_VARARGS()` 又依赖于 `OVERLOA」D_INVOKE()`，造成循环依赖。为解决此问题，将原本的重载调用功能，抽象为

CONCAT() 函数，如此一来，COUNT_VARARGS() 和 OVERLOAD_INVOKE() 就都可以基于 CONCAT() 实现[1]。

宏函数并非真的支持重载，而是设法分发调用，模拟重载。分发版本可能很多，为 OVERLOAD_INVOKE() 定义多个版本的实现，更易使用。

FOR_EACH() 就是以 OVERLOAD_INVOKE() 来调用不同的重载函数，不同的重载函数之间又利用递归思维，复用已有实现，以达到预想效果。

对于空包，在实现 COUNT_VARARGS() 时是个大问题，但此处并不会实际使用该参数，对结果没有任何影响，故不是问题。只需如此实现便可：

```
#define _FOR_EACH_0(call, ...)
```

更进一步，可以像第一章那样，以强制展开来让实现跨平台，此处省略。

使用例子：

```
void foo(int x) {
    printf("x: %d\n", x);
}

#define Foo(x) foo(x);
FOR_EACH(Foo, 0, 1, 2)

/**
 * Output:
 * x: 1
 * x: 2
 * x: 3
*/
```

这里也是为 foo() 的传递提供了一个中间件 Foo()。由于在 FOR_EACH() 的实现中，call(x) 后面并没有添加 ;，直接传递 foo 难以增加该 ;，中间件能够延迟生成，为处理提供缓冲时间。传递之时名称为 Foo，而非 Foo(arg)，仅是一个普通名称，直到将 call(x) 替换为 Foo(x) 时，才真正开始生成代码，最终替换为 foo(x);。

当然，也可以在实现中直接写成 call(x);，但那样势必会降低 FOR_EACH() 的通用性。

[1]第四章介绍的 GMP 库包含了这些修改。

2.2 Sequences

调用 `FOR_EACH(Foo, 0, 1, 2)` 时，其中含有序列，相当于 C++ 中的 `std::index_sequence<0, 1, 2>`，数量较少时还可以手动写，数量一多便异常麻烦。C++ 通过 `std::make_index_sequence<N>` 来自动生成序列，简化了写法。

我们也来提供一个宏版本的 `MAKE_INDEX_SEQUENCE(N)`, 自动生成序列。

同样利用重载加递归思想，很轻松地就能实现该需求：

```
#define MAKE_INDEX_SEQUENCE_0() 0
#define MAKE_INDEX_SEQUENCE_1() MAKE_INDEX_SEQUENCE_0(), 1
#define MAKE_INDEX_SEQUENCE_2() MAKE_INDEX_SEQUENCE_1(), 2
#define MAKE_INDEX_SEQUENCE_3() MAKE_INDEX_SEQUENCE_2(), 3
#define MAKE_INDEX_SEQUENCE_4() MAKE_INDEX_SEQUENCE_3(), 4
#define MAKE_INDEX_SEQUENCE_5() MAKE_INDEX_SEQUENCE_4()
#define MAKE_INDEX_SEQUENCE(index) \
    OVERLOAD_INVOKE(MAKE_INDEX_SEQUENCE, DEC(index))()
```

举个例子，调用 `MAKE_INDEX_SEQUENCE(5)`，将生成 `0, 1, 2, 3, 4`。实现过程中，需要调用次一版本的重载，`5` 需要减少一，代码中另外声明了 `DEC` 函数，用于实现数字自减。

`DEC()` 的实现原理也是重载，没有丝毫难度：

```
// DEC(Decrement by 1)
#define DEC_1() 0
#define DEC_2() 1
#define DEC_3() 2
#define DEC_4() 3
#define DEC_5() 4
#define DEC_6() 5
#define DEC_7() 6
#define DEC_8() 7
#define DEC_9() 8
#define DEC_10() 9
#define DEC(value) OVERLOAD_INVOKE(DEC, value)()
```

同理可以实现自增运算：

```
// INC(Increment by 1)
#define INC_0() 1
#define INC_1() 2
#define INC_2() 3
#define INC_3() 4
#define INC_4() 5
#define INC_5() 6
#define INC_6() 7
#define INC_7() 8
#define INC_8() 9
#define INC_9() 10
#define INC(value) OVERLOAD_INVOKE(INC, value)()
```

虽有 `MAKE_INDEX_SEQUENCE(N)` 生成序列，但却无法指定区间，只能生成 `[0, N)` 的序列。若想生成 `[a, b)` 的序列，如何实现呢？

2.3 Conditionals

问题的关键在于结束条件的判断，满足结束条件时，立即停止生成，否则继续生成。

先写出区间函数原型，逐步实现。

```
#define RANGE(begin, end) OVERLOAD_INVOKE(RANGE, begin)(end)
#define RANGE_0(end) 0 OVERLOAD_INVOKE( \
    RANGE_GEN_WHEN, IS_EQUAL_INT(0, DEC(end)))(1, end)
#define RANGE_1(end) 1 OVERLOAD_INVOKE( \
    RANGE_GEN_WHEN, IS_EQUAL_INT(1, DEC(end)))(2, end)
#define RANGE_2(end) 2 OVERLOAD_INVOKE( \
    RANGE_GEN_WHEN, IS_EQUAL_INT(2, DEC(end)))(3, end)
```

宏只有替换这一个功能，一切需求皆得回归到原理思考。我们需要条件判断，而已有重载函数这把称手兵器，就不是什么问题。

两个地方需要用到条件。一是当前索引是否递归到末尾，如 `[a = 0, b = 2)` 中 `a` 从 `RANGE_0()` 开始生成，若是当前索引 `a == DEC(b)`，则返回 1，否返回 0。二是根据条件一返回的结果，分发处理，1 则停止生成，0 则继续生成下一个数字。

条件一通过 `IS_EQUAL_INT(a, DEC(end))` 实现，初步代码为：

```
#define IS_EQUAL_INT_0_0() 1
#define IS_EQUAL_INT_1_1() 1
#define IS_EQUAL_INT_2_2() 1
#define IS_EQUAL_INT_3_3() 1
#define IS_EQUAL_INT_4_4() 1
#define IS_EQUAL_INT_5_5() 1
#define IS_EQUAL_INT_6_6() 1
// ...

#define IS_EQUAL_INT(a, b) \
    OVERLOAD_INVOKE(IS_EQUAL_INT, a, b)()
```

对于相等的情况，通过重载非常好写，问题在于，如何表示不相等的情况？每个数字都要比较，穷举法自然无法穷尽，此时须得另辟蹊径。

根据第一章的结论 [1.1]，对于无法解决的问题，能够通过增加一个间接层来解决。因此，考虑把结果再单独添加一个 `IS_EQUAL_INT_RESULT()` 表示，结果非 0 即 1。

如何产生非 0 即 1 的效果？在第一章中，以实现 `COUNT_VARARGS()` 展示了无数宏原理和技术，技术虽多，只取一点即够。方式就是当时所用的以 `,` 将空包变为多参的技巧，我们只需使得 `IS_EQUAL_INT_X_X()` 返回的参数个数不同，再借助 `COUNT_VARARGS()` 来分发结果便能达到目的。

怎么让返回的参数个数不等？`,` 便有此妙用。于是实现变为：

```
#define IS_EQUAL_INT_0_0() ,
#define IS_EQUAL_INT_1_1() ,
#define IS_EQUAL_INT_2_2() ,
#define IS_EQUAL_INT_3_3() ,
#define IS_EQUAL_INT_4_4() ,
#define IS_EQUAL_INT_5_5() ,
#define IS_EQUAL_INT_6_6() ,
// ...

#define IS_EQUAL_INT_RESULT_2() 1
#define IS_EQUAL_INT_RESULT_1() 0
#define IS_EQUAL_INT_RESULT(...) \
    OVERLOAD_INVOKE(IS_EQUAL_INT_RESULT, \
```

```
    COUNT_VARARGS(__VA_ARGS__))()
#define IS_EQUAL_INT(a, b) \
    IS_EQUAL_INT_RESULT( \
    OVERLOAD_INVOKE(IS_EQUAL_INT, a, b)() )
```

条件一目标达成。

2.4 Dead End

条件二通过 `RANGE_GEN_WHEN_X()` 实现，根据 `IS_EQUAL_INT()` 返回的结果调用不同的版本。

代码为：

```
#define RANGE_GEN_WHEN_1(cur, end)
#define RANGE_GEN_WHEN_0(cur, end) \
    , OVERLOAD_INVOKE(RANGE, cur)(end)

#define RANGE(begin, end) OVERLOAD_INVOKE(RANGE, begin)(end)
#define RANGE_0(end) 0 OVERLOAD_INVOKE(RANGE_GEN_WHEN, \
    IS_EQUAL_INT(0, DEC(end)))(1, end)
#define RANGE_1(end) 1 OVERLOAD_INVOKE(RANGE_GEN_WHEN, \
    IS_EQUAL_INT(1, DEC(end)))(2, end)
#define RANGE_2(end) 2 OVERLOAD_INVOKE(RANGE_GEN_WHEN, \
    IS_EQUAL_INT(2, DEC(end)))(3, end)
#define RANGE_3(end) 3 OVERLOAD_INVOKE(RANGE_GEN_WHEN, \
    IS_EQUAL_INT(3, DEC(end)))(4, end)
```

当返回结果为 1 时，表示序列生成结束，什么也不必做；当结果为 0 时，表示需要接着生成序列，于是再次调用 `RANGE_X` 的新版本。循此往复。

测试一下：

```
RANGE(0, 3)
```

想法很完美，现实很骨感。展开成了：

```
0 , 1 RANGE_GEN_WHEN_0(2, 3)
```

于是遂步分析一下，检查是否存在问题：

```
RANGE(0, 3) ->
1. OVERLOAD_INVOKE(RANGE, 0)(3)
2. RANGE_0(3)
3. 0 OVERLOAD_INVOKE(RANGE_GEN_WHEN, IS_EQUAL_INT(0,
↪  DEC(3)))(1, 3)
4. 0 OVERLOAD_INVOKE(RANGE_GEN_WHEN, IS_EQUAL_INT(0, 2))(1,
↪  3)
5. 0 OVERLOAD_INVOKE(RANGE_GEN_WHEN, 0)(1, 3)
6. 0 RANGE_GEN_WHEN_0(1, 3)
7. 0 , OVERLOAD_INVOKE(RANGE, 1)(3)
8. 0 , RANGE_1(3)
9. 0 , 1 OVERLOAD_INVOKE(RANGE_GEN_WHEN, IS_EQUAL_INT(1,
↪  DEC(3)))(2, 3)
10. 0 , 1 OVERLOAD_INVOKE(RANGE_GEN_WHEN, IS_EQUAL_INT(1,
↪  2))(2, 3)
11. 0 , 1 OVERLOAD_INVOKE(RANGE_GEN_WHEN, 0)(2, 3)
12. 0 , 1 RANGE_GEN_WHEN_0(2, 3)
```

如今停在第 12 步，换成 `RANGE(0, 1)`、`RANGE(0, 2)`、`RANGE(1, 2)`、`RANGE(1, 3)` 得到的却是正确的结果。也就是说，调用层次不能多于二层，二层之后若是调用 `RANGE_GEN_WHEN_1()`，能够得到正确的结果；若是调用 `RANGE_GEN_WHEN_0()`，就无法进一步展开。

这其实并非是思路的问题，而是宏自身的问题。宏并非是一套图灵完备的系统，它并不支持真正的递归。在二层以上，`RANGE_GEN_WHEN_0()` 又再次调用了自身，此时不会进一步展开。

这条路走死了。

2.5 Recursion

为了深入分析把路走死的原因，我们将实现进一步简化，去除多个重载版本，直接基于 `RANGE()` 递归。

```
1 #define RANGE_GEN_WHEN_1(cur, end)
2 #define RANGE_GEN_WHEN_0(cur, end) , RANGE(cur, end)
3
4 #define RANGE(begin, end) begin \
```

```
    OVERLOAD_INVOKE(RANGE_GEN_WHEN, \
    IS_EQUAL_INT(begin, end)) (INC(begin), end)
```

代码思路依旧未变，只不过前面一直采用多个函数重载来模拟递归，所以一直没有出现问题，这里直接使用递归。如果能将其实现，则前面的很多功能实现都可得到极大简化。

依旧调用 `RANGE(0, 3)`，展开结果为：

```
0 , RANGE(1, 3)
```

过程和之前相同，只是递归调用提前，替换也就提前停止：

```
RANGE(0, 3) ->
1. 0 OVERLOAD_INVOKE(RANGE_GEN_WHEN, IS_EQUAL_INT(0, 3))
↪  (INC(0), 3)
2. 0 OVERLOAD_INVOKE(RANGE_GEN_WHEN, 0) (1, 3)
3. 0 RANGE_GEN_WHEN_0 (1, 3)
4. 0 , RANGE(1, 3)
```

术语上来说，编译器扫描到 `RANGE()` 并将其展开后，便会创建一个禁用上下文（disabling context），禁用上下文会导致引用当前扩展宏的 Token 被标记为 Painted Blue，任何方式都无法再展开（这里不作深入的原理讲解，只要知晓是这个原因即可，第三章才会专门去讲这些扫描替换的具体流程和原理）。

不过禁用上下文只在一次扫描期间存在，所以可以避免引用当前扩展宏，将它延迟到下一轮扫描期间，就能够解决这个问题。延迟操作同样可以通过增加间接性来实现：

```
#define EMPTY()
#define DEFER(id) id EMPTY()

#define RANGE_GEN_WHEN_1(cur, end)
#define RANGE_GEN_WHEN_0(cur, end) \
    , DEFER(RANGE_INDIRECT)()(cur, end)

#define RANGE_INDIRECT() RANGE
#define RANGE(begin, end) begin \
    OVERLOAD_INVOKE(RANGE_GEN_WHEN, \
    IS_EQUAL_INT(begin, end)) (INC(begin), end)
```

DEFER() 的作用就是延迟宏展开。例如：

```
#define Bar() 1
Bar() --> 1
DEFER(Bar)() --> Bar()
```

由于延迟展开，Bar 在当前没有彻底展开成 1。

于是，以 DEFER() 为工具，加用 RANGE_INDIRECT() 为 RANGE() 增加一层间接性，最后生成的结果为：

```
RANGE(0, 3) ->
0 , RANGE_INDIRECT ()(1, 3)
```

这里 RANGE 不会再被标记为 Painted Blue，因此可以开启新一轮的扫描。这个操作可以通过强制展开来完成：

```
#define EXPAND(...) __VA_ARGS__

EXPAND( RANGE(0, 3) ) ->
0 , 1 , RANGE_INDIRECT ()(2, 3)

EXPAND(EXPAND( RANGE(0, 3) )) ->
0 , 1 , 2 , RANGE_INDIRECT ()(3, 3)

EXPAND(EXPAND(EXPAND( RANGE(0, 3) ))) ->
0 , 1 , 2 , 3
```

每次手动调用新一轮的扫描较为麻烦，可以预先定义多轮扫描：

```
#define EXPAND(...)  EXPAND1(EXPAND1(EXPAND1(__VA_ARGS__)))
#define EXPAND1(...) EXPAND2(EXPAND2(EXPAND2(__VA_ARGS__)))
#define EXPAND2(...) EXPAND3(EXPAND3(EXPAND3(__VA_ARGS__)))
#define EXPAND3(...) EXPAND4(EXPAND4(EXPAND4(__VA_ARGS__)))
#define EXPAND4(...) EXPAND5(EXPAND5(EXPAND5(__VA_ARGS__)))
#define EXPAND5(...) __VA_ARGS__
```

如今，调用就可以得到预想结果：

```
EXPAND( RANGE(0, 3) ) ->
0 , 1 , 2 , 3
```

这便是宏递归的实现之法，至于原理，下一章再来细论。

2.6 Examples

可以把 RANGE() 再封装一下，简化使用接口：

```
#define RANGE_IMPL(begin, end) begin \
    OVERLOAD_INVOKE(RANGE_GEN_WHEN, \
    IS_EQUAL_INT(begin, end)) (INC(begin), end)
#define RANGE(begin, end) EXPAND( RANGE_IMPL(begin, end) )

#define RANGE_INDIRECT() RANGE_IMPL
```

配合 FOR_EACH() 使用，可以批量生成序列变量，比如：

```
#define DECLARE_VARIABLES(var) int variable_ ## var;
FOR_EACH(DECLARE_VARIABLES, RANGE(2, 5))
```

将自动生成如下代码：

```
int variable_2; int variable_3; int variable_4; int
↪  variable_5;
```

同样也可以批量生成类，如：

```
#define DECLARE_CLASS(cls) class cls{};
FOR_EACH(DECLARE_CLASS, Bar, Duck)
```

生成如下代码：

```
class Foo{}; class Bar{};
```

再比如，批量生成命名空间：

```
#define START_NS(ns) namespace ns {
#define END_NS(ns) }
#define MY_NAMESPACE System, Net, Http
FOR_EACH(START_NS, MY_NAMESPACE)
int X = 42;
FOR_EACH(END_NS, MY_NAMESPACE)
```

生成如下代码：

```
namespace System { namespace Net { namespace Http {
int X = 42;
} } }
```

借助这些工具，自动生成成百上千行代码，亦不在话下。

2.7 Summary

本章以宏的核心原理进行扩展，开发 FOR_EACH 工具，迭代与递归技术穿插其中，再辅以很多小工具的实现手法，最终实现强大的代码生成组件。

灵活技巧，几近涵盖，巧妙手法，未有藏漏。

至此，大家已能够掌握在宏中编写条件、循环、递归的技术，这也是难度最高的宏元编程技巧。但是，会用不代表懂得其中的深层原理，想知道这些技术背后的 Why，便可以继续阅读第三章了。

Chapter 3

Principles of Macro Recursion

前两章主要集中于应用实践，理论概念只是蜻蜓点水，本章将重点放在宏元编程的概念原理上，深入讲解一下。

3.1 The Rescanning Rules

源文件扫描后，宏被替换为目标内容，替换实际上分为两个阶段。

第一阶段的替换发生在参数替换之时。

当宏函数含有参数时，该参数会被替换。只有两种情况例外，一是 Token 前面含有 `#` 或 `##` 预处理标记，二是后面紧跟着 `##` 预处理标记。

例如：

```
#define Def(x)          x
#define Function(x, y)  x ## y
#define Func(x)         #x
#define Fun(x)          ##x      // Not allowed
#define Fn(x)           x#       // Not allowed
#define Defn(x, y)      # x ## y // Not allowed

#define A() 1

Def(A())          // 1
Function(1, A())  // 1A()
Function(A(), 2)  // Error!
```

```
Func(A())          // "A()"
```

第二阶段的替换发生在重新扫描之时。

此时宏的参数已被第一阶段的替换展开，展开后可能会存在新的宏，为了尽可能多地寻找后续的可替换内容，将再次扫描。

例如：

```
#define Foo(x) x(x)
#define Bar(x) #x

#define A() Bar

Foo(A()) // "Bar"
```

重新扫描并非只执行一次，而是会一直循环进行，尽可能替换更多内容。比如：

```
#define Foo(x)   x(x)
#define Bar(x)   x ## 1()
#define Bar1()  hello

#define A() Bar

Foo(A()) // hello
```

需要注意，预处理器按顺序处理输入，对于之前处理过的 token，不会再作替换。例如：

```
// Example from ISO C

#define F(a) a
#define FUNC(a) (a+1)

/*
* The preprocessor works successively through the input
* without backing up through previous processed
* preprocessing tokens.
*/
F(FUNC) FUNC (3); // FUNC (3 +1);
```

此处，替换宏函数的参数列表，将 `F(FUNC)` 替换成 `FUNC`, `FUNC (3)` 替换成 `(3+1)`。由于 `FUNC` 已经被处理过，后续不会发生重新扫描，展开成 `FUNC (3+1);` 便停止处理。

具体处理步骤可表示如下：

```
/*
 * detailed:
    expand    F={ FUNC } FUNC={ (3+1) }
    remove    FUNC (3+1)

 * Final tokens output:
    FUNC (3+1)
*/
```

若非函数宏，或无参宏函数，则不会发生参数替换，直接从扫描替换开始。通过前面章节的学习，可以发现，这种循环进行的扫描替换，并非在任何时候都会进行下去。什么时候会停止呢？便是蓝染时刻。

3.2 Blue Paint

Blue Paint 是 C 预处理器中的一个黑话，一次扫描中，若是替换的 Token 引用了其自身，该 Token 将被标记为不可处理状态，这个标记动作就称为 **Painted Blue**。名称缘由已不可考，据说是由 C 工程师标记 Token 的墨水颜色而来，便延续着叫了。蓝染则是本书为称呼方便所采取的翻译，该词原为中国古代印染工艺的术语。

一个例子：

```
// Examples from ISO C

#define A  A B C
#define B  B C A
#define C  C A B

A
/*
 * detailed:
```

```
    expand    A={ A B C }
    paint     A={ a B C }
    rescan    A={ a B={ B C A } C }
    paint     A={ a B={ b C a } C }
    rescan    A={ a B={ b C={ C A B } a } C }
    paint     A={ a B={ b C={ c a b } a } C }
    remove    A={ a B={ b c a b a } C }
    remove    A={ a b c a b a C }
    rescan    A={ a b c a b a C={ C A B } }
    paint     A={ a b c a b a C={ c a B } }
    rescan    A={ a b c a b a C={ c a B={ B C A }}}
    paint     A={ a b c a b a C={ c a B={ b c a }}}
    remove    A={ a b c a b a C={ c a b c a }}
    remove    A={ a b c a b a c a b c a }
    remove    a b c a b a c a b c a

 * Final tokens output:
    A B C A B A C A B C A
*/
```

示例中，使用小写字母表示蓝染标记的 Token，一次扫描实际指的是一个 Token 的一次扫描，并非是说不会发生重新扫描。

宏预处理器按顺序逐个处理 Token，展开、替换、蓝染、移除……因为 ABC 相互引用，当所有 Token 都被处理完成之后，它们也全部被蓝染标记。

再看一个例子：

```
#define Foo(X) 1 Bar
#define Bar(X) 2 Foo

Foo(X)(Y)(Z)

/*
 * detailed:
    expand    Foo={ 1 Bar }(Y)(Z)
    remove    1 Bar(Y)(Z)
    rescan    1 Bar={ 2 Foo }(Z)
```

```
    remove    1 2 Foo(Z)
    rescan    1 2 Foo={ 1 Bar }
    remove    1 2 1 Bar


  * Final tokens output:
    1 2 1 Bar
*/
```

此处，虽为宏函数，但参数却形同虚设，没有发生参数替换，也没有发生蓝染标记，就是不断扫描、替换、再扫描……重复进行，直到无可替换。

3.3 Blue Paint and Recursion

蓝染是 C 预处理器不支持递归的原因，知道了这个原理，再去看第二章的内容，便能够剖析入微，鞭辟入里。

这里摘出来一个简洁版本讲解：

```
#define RECURSIVE(x) x RECURSIVE(x)

RECURSIVE(1)

/*
 * detailed:
    expand    RECURSIVE={ 1 RECURSIVE(1) }
    paint     RECURSIVE={ 1 recursive(1) }
    remove    1 recursive(1)

 * Final token output:
    1 RECURSIVE(1)
*/
```

`RECURSIVE(1)` 处理完成之后被蓝染，致递归还未开始，便已终止。

蓝染后的 Token，即便是强制展开，也不会有半点效果：

```
#define EVAL1(...) __VA_ARGS__
#define RECURSIVE(x) x RECURSIVE(x)
```

```
3
4  // Not working
5  EVAL1(RECURSIVE(1))
6
7  /*
8   * detailed:
9      expand    EVAL1( RECURSIVE={ 1 RECURSIVE(1) } )
10     paint     EVAL1( RECURSIVE={ 1 recursive(1) } )
11     remove    EVAL1( 1 recursive(1) )
12     rescan    1 recursive(1)
13
14  * Final token output:
15     1 RECURSIVE(1)
16 */
```

因为重新扫描会循环发生，所以即使你增加一个间接层，也不会对结果产生丝毫影响：

```
1  #define EVAL1(...) __VA_ARGS__
2  #define RECURSIVE(x) x _RECURSIVE()(x)
3  #define _RECURSIVE() RECURSIVE
4
5  // Not working
6  EVAL1(RECURSIVE(1))
7
8  /*
9   * detailed:
10     expand    EVAL1( RECURSIVE={ 1 _RECURSIVE()(1) } )
11     rescan    EVAL1( RECURSIVE={ 1 RECURSIVE(1) } )
12     paint     EVAL1( RECURSIVE={ 1 recursive(1) } )
13     remove    EVAL1( 1 recursive(1) )
14     rescan    1 recursive(1)
15
16  * Final token output:
17     1 RECURSIVE(1)
18 */
```

因此，首先需要一种方法，能够禁止重新扫描。这种方法就是延迟扫描，基本代码逻辑如下所示：

```
#define EMPTY()
#define DEFER(id) id EMPTY()

#define Bar() 1
Bar() // 1
DEFER(Bar)() // Bar ()

/*
 * DEFER(Bar)()
 * detailed:
    expand    Bar EMPTY()()
    rescan    Bar ()
*/
```

第一节说过，预处理器按顺序处理输入，对于之前处理过的 Token，不会再作处理。`Bar` 便是已经处理过的 Token，不会再对它扫描。

将这个技术应用到递归代码中，便可以避免蓝染标记。

```
#define EMPTY()
#define DEFER(id) id EMPTY()
#define EVAL1(...) __VA_ARGS__
#define RECURSIVE(x) x DEFER(_RECURSIVE) ()(x)
#define _RECURSIVE() RECURSIVE

EVAL1(RECURSIVE(1))

/*
 * detailed:
    expand  EVAL1( RECURSIVE={ 1 DEFER(_RECURSIVE) ()(1) } )
    rescan  EVAL1( RECURSIVE={ 1 _RECURSIVE ()(1) } )
    remove  EVAL1( 1 _RECURSIVE ()(1) )
    rescan  EVAL1={ 1 RECURSIVE(1) ) }
    rescan  EVAL1={ 1 RECURSIVE={1 DEFER(_RECURSIVE) ()(1) }
↪   }
```

```
    rescan  EVAL1={ 1 RECURSIVE={1 _RECURSIVE ()(1) } }
    remove  EVAL1={ 1 1 _RECURSIVE ()(1) }
    remove  1 1 _RECURSIVE ()(1)

 * Final token output:
    1 1 _RECURSIVE ()(1)
*/
```

第二节说过，蓝染标记只在一次扫描期间存在，通过间接性加上延迟扫描，`RECURSIVE` 便永远不会再被蓝染标记。但是，禁用重新扫描，我们也会因此失去循环扫描的能力，必须手动通过 `EVAL1` 来执行一次扫描。

手动调用扫描的次数和递归的深度成正比，一次次手动太过麻烦，故而一般会提前定义好多层扫描，以便后续使用。

```
#define EVAL(...)  EVAL1(EVAL1(EVAL1(__VA_ARGS__)))
#define EVAL1(...) EVAL2(EVAL2(EVAL2(__VA_ARGS__)))
#define EVAL2(...) EVAL3(EVAL3(EVAL3(__VA_ARGS__)))
#define EVAL3(...) EVAL4(EVAL4(EVAL4(__VA_ARGS__)))
#define EVAL4(...) EVAL5(EVAL5(EVAL5(__VA_ARGS__)))
#define EVAL5(...) __VA_ARGS__
```

有了该工具，便可基本模拟宏递归，本章只是穿插着讲原理，更多使用见第二章。

3.4 Summary

总的来说，宏预处理器的替换规则算不上晦涩，但相关资料寥寥无几，文章更是少得可怜，像是黑盒子，导致理解起来并不容易。

本章对此进行了深入剖析，篇幅不长，主要是应用放在了第二章，应用加上原理，总算是覆盖了宏元编译的核心知识点。

整体上而言，宏元编程很是复杂，这几章具有一定的难度。

Chapter 4

The GMP Library

宏部分的核心理论和技术，于前三章，悉已更讫。利用这些原理和技巧，可以实现一些简单的代码生成工具，得到初步的产生式元编程能力。

本书共有十章，宏元编程只是其中的第一个模块，属于上篇。上篇完结后，将产生一个产生式元编程库，再步入中篇。由是，本系列由虚向实，自理论技术诞生项目，复从项目推进续篇，循此迭代。

因此，本章乃承上启下之篇，为上篇画句号，为中篇起草稿。

4.1 The Generative Metaprogramming Library

本书将同步产生一个产生式元编程库，名为 GMP[1]。该库的侧重点是提供产生式编程工具，可在编译期实现代码生成。

基于前三章的内容，已添加宏模块的基本功能，跨平台，可实现简单的代码生成。其余模块将随着后续章节持续添加。

下面是一些使用例子。

获取可变宏参数数量：

```
#include <gmp/gmp.hpp>

int main() {
    // Output: 0 1 2 3 4 5 6 7
    printf("%d %d %d %d %d %d %d %d\n",
        GMP_SIZE_OF_VAARGS(),
```

[1] https://github.com/lkimuk/gmp

```
        GMP_SIZE_OF_VAARGS(1),
        GMP_SIZE_OF_VAARGS('a', 'b'),
        GMP_SIZE_OF_VAARGS('a', 'b', 'c'),
        GMP_SIZE_OF_VAARGS('a', 'b', 1, 2),
        GMP_SIZE_OF_VAARGS('a', 'b', 1, 2, 3),
        GMP_SIZE_OF_VAARGS('a', 'b', 1, 2, 3, 4),
        GMP_SIZE_OF_VAARGS('a', 'b', 1, 2, 3, 4, 5));
}
```

生成下标索引：

```
#include <gmp/gmp.hpp>

int main() {
    // Output: 0 1 2 3 4 5 6 7 8 9
    printf("%d %d %d %d %d %d %d %d %d %d\n",
        GMP_MAKE_INDEX_SEQUENCE(10));
}
```

生成范围索引：

```
#include <gmp/gmp.hpp>

int main() {
    // Output: 10 11 12 13 14 15 16 17 18 19
    printf("%d %d %d %d %d %d %d %d %d %d\n",
        GMP_RANGE(10, 20));
}
```

批量生成变量：

```
#include <gmp/gmp.hpp>

#define DECLARE_VARIABLES(v) int variable_ ## v;

int main() {
    GMP_FOR_EACH(DECLARE_VARIABLES, GMP_RANGE(1, 4))

```

```
    variable_1 = 10;
    variable_2 = 20;
    variable_3 = 30;

    // Output: 10 20 30
    printf("%d %d %d\n", variable_1, variable_2, variable_3);
}
```

批量生成不变操作：

```
#include <gmp/gmp.hpp>

void bar(int arg1, const char* arg2) {
    printf("bar: %d, %s\n", arg1, arg2);
}

#define GENERATE_SOMETHING(arg1, arg2) bar(arg1, arg2);

int main() {
    GMP_LOOP(GENERATE_SOMETHING, 10, 42, "arg2")
}

// Output:
// bar: 42, arg2
// bar: 42, arg2
// bar: 42, arg2
// bar: 42, arg2
// bar: 42, arg2
// bar: 42, arg2
// bar: 42, arg2
// bar: 42, arg2
// bar: 42, arg2
// bar: 42, arg2
```

基于现有功能，已能轻松实现许多代码生成需求；如果有更加复杂的产生式需求，宏不太合适，可考虑通过其他模块来提供支持。

4.2 Features

GMP 中的宏模块，主要包含前三章所讲解的工具，但是要更加完善、易用。

所有功能都以产生式需求为导向，而非是为封装完善的预处理功能（如 `BOOST_PP` 预处理库），因此要更加精简且具有针对性。若是超级复杂的产生式需求，采用宏的奇技淫巧反而会使问题更加复杂，语法也难以直观。因此，复杂的产生式需求，不会包含在宏模块当中。

宏模块包含的工具大体可分为：可变宏参数大小、宏拼接、宏重载、宏循环、宏递归、宏位操作、宏自增自减、宏布尔转换、宏条件逻辑、宏数值比较。

没有需求、或是其他模块能够更好地支持的功能，暂不会放在宏模块当中。

4.3 Limitations and Cross-Platform Compatibility

宏并非是图灵完备的系统，具有诸多限制，这是首先需要考虑的问题。

宏嵌套的最大深度不能超过 256 层，否则会产生编译错误，因此，一切索引都应该小于 256 层。这并非大问题，宏只是用来完成简单的代码生成，256 层递归完全可以满足基本需求。这样也为宏代码明确设置了一个上限，减少代码的行数。

虽然说最多有 256 层，但是辅助宏也会占用一定的层数，于是 GMP 的实际最大索引值是 254。

此外，GMP 不再使用 C++20 的相关特性，如 `__VA_OPT__`。因为 MSVC 默认不更新 `__cplusplus`，该值一直是 `199711`，需要使用 `/Zc:__cplusplus` 开启才会变化。这样就没必要像第一章那样分成 C++20 和之前的实现方式，统一用之前的实现方式更加简洁。

需要注意，MSVC 默认使用的是非标准宏，还需要使用 `/Zc:preprocessor` 开启标准宏，否则宏的行为和 C++ 标准不一致。

4.4 Macro Error Handling

报错对于宏来说也是非常关键的一个功能，没有报错，就无法检测索引是否超出最大限制，执行逻辑可能会产生无法预料的行为。

GMP 实现 `GMP_CHECK_INDEX(index)` 和 `GMP_CHECK_INDEX_BOOL(index)` 来检查错误。前者将直接产生错误信息，后者将返回一个布尔值。

例如，调用如下操作：

```
GMP_MAKE_INDEX_SEQUENCE(255);
```

将存在错误，因为 GMP 的最大索引值是 254。不过不会发生未定义行为，GMP 通过前两个错误宏工具，使编译器可以产生错误。比如，使用 Clang 编译，将产生以下错误：

```
error: use of undeclared identifier
↪  'Error_Index_255_Exceeds_Maximum_Macro_Index_254'
```

其他编译器也有相似的提示。这不仅在发生错误时能够中止编译，还提供了顾名思义的错误信息。

4.5 Looking Ahead

可以说，本书各章的内容既是教程，也是 GMP 的开发文档，讲解了所有关键技术，而上篇所对应的，就是其中的宏模块。

原想一边更新后续章节，一边完善 GMP 库，但为尘事所羁，能写完本书已是不易，遑论写库。

本书如有第二版，会再行完善本章。

Chapter 5

Basics of Templates

宏是 C 时期的产物，功能颇为简陋，也不是图灵完备的，即使诸般殊法诡技加身，限制依旧很多，且调试不易，有所束缚，以是仅作辅助，用来实现简单的代码生成功能。至 C++ 时期，产生式元编程工具陡增，其中以模板为一切的根基，发枝散叶，如今正向静态反射前进。

本章主要集中于模板的核心理论和用法，概念错综复杂，技术层出不穷，但不会涉及模板的细枝末节。难度绝对不低，是非常重要的一章。

纵使某些概念和技术你已知悉，也莫要跳过某个小节，因为本书的重心不在模板编程，而在产生式编程，论述角度将有所差异。

5.1 Generic Programming and Metaprogramming

传统过程式编程的思路是将逻辑作为函数，数据作为函数的输入和输出，通过无数个函数来组织完整的逻辑需求，好似搭积木，构建出各种物体。

面向对象编程更进一步，思路是将逻辑和数据合并起来，构成一个个类，类中包含各种数据和函数，外部只能借助公开函数操纵这些数据。这种以对象表示现实事物的方式与人的思维更加接近，简化了抽象化的难度。

泛型编程则欲将逻辑和数据分离，并提供一种抽象化数据的方式，使得不同的数据能够使用同一种逻辑，极大降低了代码的重复性。于是，同一个函数能够传递不同的参数类型，这就是函数模板；同一个类能够传递不同的数据，这就是类模板。

C++ 是一种多范式语言，以模板支持泛型编程，允许编写泛化的函数和数据，功能不依赖于某种具体的数据类型，使用 SFINAE 或 Concepts 约束抽象化

的类型。这种能力使其支持 Parametric Polymorphism，在多态的选择上不必局限于传统的 Subtype Polymorphism，可以使用编译期多态替代运行期多态。

C++ 中，元编程指的是发生于编译期的编程，模板为 C++ 带来泛型编程的同时，也带来了元编程能力。由模板实现的元编程，称为模板元编程。而产生式元编程，指的是发生于编译期的对于编程的编程，模板本就支持在编译期生成代码，因此模板也能够实现产生式元编程。

5.2 Abstraction, Concretization, and Templates

在模板世界中，函数不再是具体的函数，数据类型也不再是具体的数据类型。模板宛如一个模具，借其能够产生各式各样的物体，这些物体的颜色、材料可能不尽相同，但是形状、大小是一致的。也就是说，一类物体，必须存在共同的部分，才能够抽象出一个模具，模具的作用就是重复利用这些共同之处，减少重复性。

编程是工具，本质是解决问题，解决问题考验的就是抽象化和具体化的能力。抽象化应对的是现实世界中的不变，而具体化应对的是现实世界中的变化，能够清楚地认识到变与不变是什么，问题也就迎刃而解了。那先来分清抽象化和具体化的概念。

抽象的意思是，在许多事物中，去除非本质的属性，抽出本质属性。抽象化就是呈现出具体事物共同本质的过程。将复杂的现实，简化成单纯的模型，这种抽象化方式称为模型化，所谓问题建模，就是对问题进行模型化，也即抽象的过程。具体的概念相对简单，就是指看得见、摸得着的事物，每个事物都是独一无二的，是以变化尤盛。具体化就是将抽象事物加载清晰的过程，是一种有助于理解陌生事物的方式。数据类型是具体事物，能够清晰简明地呈现出不同数据类型的共同逻辑，就是用模板类型代替具体类型的精妙之处。

模板本身是一种抽象化的工具，其编写的是抽象逻辑，表示某类类型的共有本质，是不变的部分。其本身是没有用的，实际需要的还是具体逻辑，因此需要有具体化的过程，将抽象逻辑转变为具体逻辑。在 C++ 模板中，这种将抽象逻辑转变成具体逻辑的过程，称为实例化。实例化将在编译期把抽象类型替换成具体类型，根据模板生成一个个具体逻辑。这种实例化机制，便是我们所需要的代码生成能力。

与宏不同，模板具体化后生成的代码，并无法直接在生成的源码中看到，不过可以通过 CppInsight 这类工具察看。比如：

```
1 template<typename T>
2 inline constexpr T Integer = 42;
```

```
3
4  int main() {
5      std::cout << Integer<int> << Integer<long> << "\n";
6  }
```

将在编译期生成以下代码：

```
1  template<typename T>
2  inline constexpr const T Integer = 42;
3
4  template<>
5  inline constexpr const int Integer<int> = 42;
6  template<>
7  inline constexpr const long Integer<long> = 42;
```

由此，通过模板，我们拥有了生成数据和函数的能力。

5.3 Variable, Function, and Class Templates

在编程语言中，能够操纵的最小单元无非就是数据和函数这两类。数据构成了变量及函数的输入输出，数据和函数又组成了类，抽象化作用其上，便可分成变量模板、函数模板和类模板。

5.3.1 Variable Templates

变量模板是针对变量的抽象化，与其他模板不同，它直到 C++14 才进入标准。

囿于 C++98/03 的局限性，旧时无法直接对变量施加模板。于是，产生了一些替代之法，一种是在类模板中使用 `constexpr static` 数据成员，另一种是通过 `constexpr` 函数模板返回常量值。

第一种方法倒是经典，比如 `std::numeric_limits` 的实现：

```
1  template<typename T>
2  struct numeric_limits {
3      static constexpr bool is_modulo = ...;
4  };
5
```

```
// ...
template<typename T>
constexpr bool numeric_limits<T>::is_modulo;
```

这种方法存在两个缺点，一是重复性，必须在类外定义静态数据成员，以满足 ODR-used 数据（这点在 C++17 之后不再是问题，通过 `inline` 可以光明正大地违背 ODR）；二是简洁性，使用时需要通过 `numeric_limits<X>::is_modulo` 这种冗余的语法形式。

第二种方法，同样可以看 `std::numeric_limits` 的实现：

```
template<>
struct numeric_limits<int>
{
    static constexpr int
    min() noexcept { return -__INT_MAX__ - 1; }

    static constexpr int
    max() noexcept { return __INT_MAX__; }

    ...
};
```

这种方法不存在第一种方法当中的重复性问题，但在定义时需要提前选择如何传递常量。要么通过 `const` 引用传递，此时需要返回一个存储于静态区常量；要么通过值传递，每次返回时都复制一份常量。直接使用 `const(expr)` 不会存在这个问题，常量是否需要实际存储只依赖于使用情境，而不依赖于定义。

事实上，编译器的优化能力今非昔比，RVO 在某种程度上也能够避免这些缺陷，但毫无疑问，变量模板能够更直接、简单、高效地解决问题。

后期的实现都会避免这些替代方法，而是直接采用变量模板来解决相关问题。于是，可以看到 `xxx::value` 都被 `xxx_v` 进一步替代，Mathematical constants 也清一色地使用 `inline constexpr` 变量模板：

```
namespace std::numbers {
    template<class T> inline constexpr T e_v           = /*
    ↪ unspecified */;
    template<class T> inline constexpr T log2e_v       = /*
    ↪ unspecified */;
```

```
4     template<class T> inline constexpr T log10e_v     = /*
  ↪   unspecified */;
5     template<class T> inline constexpr T pi_v         = /*
  ↪   unspecified */;
6     template<class T> inline constexpr T inv_pi_v     = /*
  ↪   unspecified */;
7     template<class T> inline constexpr T inv_sqrtpi_v = /*
  ↪   unspecified */;
8     template<class T> inline constexpr T ln2_v        = /*
  ↪   unspecified */;
9     template<class T> inline constexpr T ln10_v       = /*
  ↪   unspecified */;
10    template<class T> inline constexpr T sqrt2_v      = /*
  ↪   unspecified */;
11    template<class T> inline constexpr T sqrt3_v      = /*
  ↪   unspecified */;
12    template<class T> inline constexpr T inv_sqrt3_v  = /*
  ↪   unspecified */;
13    template<class T> inline constexpr T egamma_v     = /*
  ↪   unspecified */;
14    template<class T> inline constexpr T phi_v        = /*
  ↪   unspecified */;
15
16    inline constexpr double e          = e_v<double>;
17    inline constexpr double log2e      = log2e_v<double>;
18    inline constexpr double log10e     = log10e_v<double>;
19    inline constexpr double pi         = pi_v<double>;
20    inline constexpr double inv_pi     = inv_pi_v<double>;
21    inline constexpr double inv_sqrtpi =
  ↪   inv_sqrtpi_v<double>;
22    inline constexpr double ln2        = ln2_v<double>;
23    inline constexpr double ln10       = ln10_v<double>;
24    inline constexpr double sqrt2      = sqrt2_v<double>;
25    inline constexpr double sqrt3      = sqrt3_v<double>;
26    inline constexpr double inv_sqrt3  = inv_sqrt3_v<double>;
```

```
27      inline constexpr double egamma     = egamma_v<double>;
28      inline constexpr double phi        = phi_v<double>;
29  }
```

如今，若是想使用常量 π，可以直接通过 `std::numbers::pi` 得到默认类型的，显式地传递模板参数，可以得到不同类型的常量。

注意，变量模板常在 file scope 下使用，此时 `constexpr` 会隐式 Internal Linkage，倘只单独使用 `constexpr` 修饰变量模板，多个 TUs 间链接会拷贝多份数据，造成巨大的开销。而 `inline` 能够指定 External Linkage，避免这种开销，因此，变量模板一般都采用 `inline constexpr` 修饰。

这些都只是标准中的例子，现实中有哪些用到变量模板的例子呢？

第一个例子，假设你在开发某个模拟器，需要指定模拟的开始时间和结束时间，模拟器是全局可用的，其中的所有模拟设备都会使用该模拟时间。此时的一种做法就是将时间以 `inline constexpr/constinit` 变量模板放到全局配置，从而能够分别指定和获取以 `milliseconds/seconds/minutes/hours...` 为精度的模拟时间。当然，更好的做法是像 ASIO 那样，抽象出一个 `sim_context`，将所有配置内容放到该类当中，之后再将该上下文对象传递到每个模拟设备当中，从而避免全局对象。

第二个例子，变量模板可以结合 Lambda 重载使用。泛型 Lambda 虽然具有函数模板和类模板的部分特征，但它的数据部分只能通过捕获参数，无法真正像类那样使用。通过结合变量模板，泛型 Lambda 得以像类那样传递数据类型。具体的例子，这里引用泛型 *Lambda，如此强大*[1] 中展示的抽象工厂法：

```
1   template<class... Ts>
2   struct AbstractAIFactory
3       : Ts... { using Ts::operator()...; };
4   template<class... Ts> AbstractAIFactory(Ts...)
5       -> AbstractAIFactory<Ts...>;
6
7   template<class T, class U>
8   concept IsAbstractAI = std::same_as<T, U>;
9
10  template<class T>
11  static constexpr auto AIFactory = AbstractAIFactory {
12      []() requires IsAbstractAI<T, Lux> {
```

[1] https://www.cppmore.com/2022/01/16/generic-lambda

```
        return new LuxEasy; },
    []() requires IsAbstractAI<T, Ziggs> {
        return new ZiggsEasy; },
    []() requires IsAbstractAI<T, Teemo> {
        return new TeemoEasy; }
};

const auto lux = AIFactory<Ziggs>();
lux->print();
```

此处 `AIFactory` 并不位于 file scope，于是可以使用 `static constexpr` 强保证编译期执行。这里的重复性模板无法解决，但是借助 GMP 库目前的代码生成能力，解决起来却是不再话下。可以简化为：

```
#define CONCRETE_AI_FACTORY(x) \
    []() requires IsAbstractAI<T, x> { \
        return new GMP_CONCAT(x, Easy); },

template<class T>
static constexpr auto AIFactory = AbstractAIFactory {
    GMP_FOR_EACH(CONCRETE_AI_FACTORY, Lux, Ziggs, Teemo)
}
```

这也是结合不同类型产生式工具的一个例子。

至此，关于变量模板的讨论告一段落，继续来看下一类模板。

5.3.2 Function Templates

函数模板是针对函数的抽象化，借此可产生具有同一逻辑的函数族，C++98/03 便进入标准。

函数模板是抽象化的函数表达，抽象之物本身并不存在，它既不是类型，也不是函数。具体化之后的代码才拥有实体，具有真正的参数类型，才属于函数。生成可用代码的过程，称为函数模板实例化，实例化后生成的函数称为模板函数。

函数模板允许编写泛型函数，比如：

```
template<class T>
const T& min(const T& a, const T& b) {
```

```
    return b < a ? b : a;
}

template<class T>
const T& max(const T& a, const T& b) {
    return b < a ? a : b;
}
```

`min()` 和 `max()` 可以接受支持 `operator<` 的操作数，如此便抽象了所有类型的最大最小逻辑，无需为每个类型重复编写相同的功能。

注意，这里 `max()` 的实现与标准的实现不同，标准采用的是 `a < b ? b : a`。在两个操作数等价却不相等时，标准实现将返回 `a`，而此处的实现将返回 `b`。采取这种实现的目的是保证更强的一致性，例如 `{min(a, b), max(a, b)}`，在`a` 和 `b` 等价而不相等时，标准实现将返回 `{a, a}`，此处实现将返回 `{a, b}`，一个置小，另一个便置大，更加利于排序。

C++ 中等价的定义是 `!(a < b) && !(b < a)`，而相等的定义是 `a == b`，前者只对比部分 values，而后者对比全部 values 和 types。下面是一个完整的示例：

```
// Check if the two operands are equivalent.
template<class T>
constexpr bool equivalent(T const& a, T const& b) {
    return !(a < b) && !(b < a);
}

// Check if the two operands are equal.
template<class T>
constexpr bool equal(const T& a, const T& b) {
    return a == b;
}

// max() in standard
template<class T>
constexpr const T& max1(T const& a, T const& b) {
    return a < b ? b : a;
}

```

```
// max() in example
template<class T>
constexpr const T& max2(T const& a, T const& b) {
    return b < a ? a : b;
}

struct X { int a, b; };
constexpr bool operator==(X const& lhs, X const& rhs) {
    return lhs.a == rhs.a && lhs.b == rhs.b;
}

constexpr bool operator<(X const& lhs, X const& rhs) {
    return lhs.a < rhs.a;
}

int main() {
    constexpr X x1{ 0, 1 };
    constexpr X x2{ 0, 2 };

    static_assert(!equal(x1, x2),
        "x1 and x2 are equal!");
    static_assert(equivalent(x1, x2),
        "x1 and x2 are not equivalent!");

    static_assert(x1 == max1(x1, x2),
        "x1 was not returned by max1");
    static_assert(x2 == max2(x1, x2),
        "x2 was not returned by max2");
}
```

函数模板亦可重载，不过在重载决议中非模板函数要优于模板函数，此类话题，不再细述，见 *The Book of Modern C++*/p63[2]。

[2] https://github.com/lkimuk/the-book-of-modern-cpp

5.3.3 Class Templates

类模板是针对类的抽象化，借此可产生不同类型的数据和函数，支持偏特化，用处最广。C++98 即存在，至 C++03 各类奇技淫巧已遍地生花，极一时之盛，啧啧之声不绝。

类模板与前两类模板最大的不同在于类能够继承，这其中最绝妙的技术是递归继承。最经典的应用是 `std::tuple` 的实现，下面展示精简版的一种实现：

```
template<typename... Types> class Tuple;
template<> class Tuple<> {};

template<typename Head, typename... Tail>
class Tuple<Head, Tail...>
    : public Tuple<Tail...> {
    using TailType = Tuple<Tail...>;
protected:
    Head head_;

public:
    Tuple() {}
    Tuple(Head v, Tail... vtails) : head_(v),
    ↪ TailType(vtails...) {}

    Head head() { return head_; }
    TailType& tail() { return *this; }
};
```

递归继承将依次继承可变模板参数，直至 `Tuple<>`，假如调用为 `Tuplet(1, 2.0, "blah blah...")`，展开后如下所示：

```
------------------------------
Tuple<>                      |
------------------------------
↑
------------------------------
Tuple<string>                |
string head_("blah blah...");|
```

```
------------------------------
↑
------------------------------
Tuple<float, string>         |
float head_(2.0);            |
------------------------------
↑
------------------------------
Tuple<int, float, string>    |
int head_(1);                |
------------------------------
```

递归继承是模板当中最强的代码生成技术，寥寥数行实现，便能自动产生成千上万行代码。相应地，这部分运用起来极为复杂，下一章将单独讨论。

5.4 A Deep Dive into Explicit Template Instantiation

模板实例化的形式分为两种，一种称为显式实例化，一种称为隐式实例化。模板实例化的变量、函数和类等实体叫作特化。

显式实例化语法具有两种形式，显式实例化定义和显式实例化声明，如下所示：

```
// explicit instantiation definition
template declaration

// explicit instantiation declaration
extern template declaration
```

显式实例化不能添加 `inline/constexpr/consteval` 修饰符，也不能添加属性修饰。

对于较少接触模板的 C++ 开发者来说，可能从未使用过显式实例化。但对于产生式元编程来说，模板运用频率极高，这无疑会增加编译时间和输出大小，而显式实例化能够优化这方面的开销。

若要深入讨论这部分内容，需要先理解隐式实例化，否则难以知晓开销从何而来。

隐式实例化就是编译器将抽象模板转换成具体代码的方式，编译器和链接器必须确保每个模板实例在可执行程序中只会存在一份。C++ 标准并没有规定

到底应该采用哪种方式实现，事实上，主要存在两种基本的解决方案，分别是 Borland 模型和 Cfront 模型。

Borland 模型中，编译器在每个使用模板的翻译单元中都会生成模板实例，再由链接器将多个翻译单元合并起来，剔除重复生成的模板实例，只保留一份。此模板只需考虑目标文件本身，无需担心外部复杂性。但是，由于重复生成多个模板实例，模板代码被重复编译，导致编译时间增长。这种模型下，模板代码的定义通常需要放在头文件当中，以在实例化时能被找到。

Cfront 模型中，增加了一个模板库的概念，用来自动维护模板实例。当构建单个目标文件时，编译器会将遇到的模板定义和实例化放入模板库，如果实例化已经存在于模板库中，编译器会重用该实例化，而不是重新生成。在链接时，链接包装器会添加模板库中的对象，并编译任何之前未生成的所需实例。这个模型对编译速度的优化程度更高，并且无需像 Borland 模型那样需要替换链接器，直接使用系统链接器即可。但是，复杂性也急剧增加，出错的概率更大。实践中，在一个目录中构建多个程序和在多个目录中构建一个程序可能会非常困难。这种模型下，代码通常会将非内联成员模板的定义分离出来，放到一个文件中单独编译。

GNU C++ 采用的便是 Borland 模型，在编译模板代码时，每个翻译单元中都会生成一份使用的模板实例，随后再由链接器移除重复生成的实例，这便是导致编译时间和输出大小增加的罪魁祸首。

如何干掉这个罪魁祸首？

这里不谈某个编译器特有的解决方式，只说 C++ 标准所提供的通用方式，即本节开头所说的显式实例化。

显式实例化定义 C++03 就已支持，主要是用来强制编译器生成实例化模板；显式实例化声明于 C++11 加入标准，主要是用来禁止编译器在翻译单元中实例化模板。换言之，后者能够阻止隐式实例化。阻止之后便避免了 Borland 模型带来的编译开销，减少编译时间，那如何再使用模板实例化呢？就是通过显式实例化定义，手动强制编译器生成。

口说无凭，我们先来验证一下编译器是否真的会生成多个模板实例。

先准备模板头文件 `foo.h`，内容为：

```
1 // foo.h
2 #ifndef FOO_H_
3 #define FOO_H_
4
5 #include <iostream>
6
```

```
template<class T>
class Foo {
public:
    Foo(T x) : data_(x) {}

    void print() const {
        std::cout << "x:" << data_ << "\n";
    }

private:
    T data_;
};

#endif // FOO_H_
```

再准备两个源文件，其中使用该模板定义：

```
// source_one.cpp
#include "foo.h"

void do_something1() {
    Foo<int> foo(1);
    foo.print();
}

// source_two.cpp
#include "foo.h"

void do_something2() {
    Foo<int> foo(2);
    foo.print();
}
```

分别使用如下命令，查看两个翻译单元的符号表：

```
nm -C -S source_one.o | grep Foo
nm -C -S source_two.o | grep Foo
```

符号表中的内容分别为：

```
// source_one.o
0000000000000000 000000000000001b W Foo<int>::Foo(int)
0000000000000000 000000000000001b W Foo<int>::Foo(int)
0000000000000000 n Foo<int>::Foo(int)
0000000000000000 0000000000000048 W Foo<int>::print() const

// source_two.o
0000000000000000 000000000000001b W Foo<int>::Foo(int)
0000000000000000 000000000000001b W Foo<int>::Foo(int)
0000000000000000 n Foo<int>::Foo(int)
0000000000000000 0000000000000048 W Foo<int>::print() const
```

第一列表示符号在目标文件中的地址，`0000000000000000` 表示编译器在编译阶段没有为它们分配实际地址，这些地址将在链接阶段由链接器进行分配和处理；第二列表示生成代码的大小；第三列表示符号类型，`W` 表示弱符号，弱符号允许多个定义在链接时共存，`n` 表示局部符号，是编译器生成供内部使用的符号，不会被导出到最终的链接阶段；最后一列就是符号的名称。

同一个对象文件中出现了相同的构造函数符号，这是为什么呢？直接输出 Mangled Name 查看一下：

```
0000000000000000 W _ZN3FooIiEC1Ei
0000000000000000 W _ZN3FooIiEC2Ei
0000000000000000 n _ZN3FooIiEC5Ei
0000000000000000 W _ZNK3FooIiE5printEv
```

可见，尽管 Demangled Name 是相同的，但 Mangled Name 却不相同。唯一的区别在于 `C1/C2`，其意义为：

```
<ctor-dtor-name> ::= C1 # complete object constructor
        ::= C2          # base object constructor
        ::= C3          # complete object allocating
        ↪  constructor
        ::= D0          # deleting destructor
        ::= D1          # complete object destructor
        ::= D2          # base object destructor
```

C2 表示基类对象构造，例子中并不涉及继承，不需要生成这个符号，但编译器依然生成了。GCC Bugs[3] 中提及，尽管该问题频繁被报告，但并不是实际的错误：

> G++ emits two copies of constructors and destructors.
>
> In general there are three types of constructors (and destructors).
>
> 1. The complete object constructor/destructor.
> 2. The base object constructor/destructor.
> 3. The allocating constructor/deallocating destructor.
>
> The first two are different, when virtual base classes are involved.

例子中没有虚基类，C1/C2 是相同的，在足够的优化级别上，GCC 实际上会将两个符号链接到相同的代码。因此，不必太过在意出现相同的构造函数符号，只看一个就行。

两个翻译单元，source_one.o 和 source_two.o 之中，为每个模板函数都生成了一个单独的节，并占据着一定的空间。若是不计其数的翻译单元中都使用该模板，编译器会为每个单元都实例化模板，增加编译时间，重复占用输出文件的大小。

显式实例化如何避免这种开销？有三种方式，下面分别展示。

第一种，将模板定义从头文件移动到 CPP 源文件，头文件中只留下模板声明，并在源文件中使用显式实例化定义强制编译器生成模板实例。

也就是说，将代码变为：

```
1  // foo.h
2  #ifndef FOO_H_
3  #define FOO_H_
4
5  template<class T>
6  class Foo {
7  public:
8      Foo(T x);
9
10     void print();
11
12 private:
```

[3] http://gcc.gnu.org/bugs/#known

```
    T data_;
};

#endif // FOO_H_

// foo.cpp
#include "foo.h"
#include <iostream>

template<class T>
Foo<T>::Foo(T x) : data_(x) {}

template<class T>
void Foo<T>::print() {
    std::cout << "x:" << data_ << "\n";
}

// Explicit template instantiate
template class Foo<int>;
```

此时再来观察 source_one.cpp 和 source_two.cpp 生成目标当中的符号表，如下所示：

```
// source_one.o
                U Foo<int>::print()
                U Foo<int>::Foo(int)

// source_two.o
                U Foo<int>::print()
                U Foo<int>::Foo(int)
```

U 表示未定义的符号，模板定义如今放到了 CPP 源文件，这两个翻译单元无法看见，自然不会编译，也没有占用目标文件大小。唯一的一份模板实例在 foo.cpp 当中，它们可以使用此份实例，链接之时并不会存在问题。

添加一个 main.cpp 提供主函数：

```
extern void do_something1();
extern void do_something2();
```

```

int main() {
    do_something1();
    do_something2();
}
```

链接输出如下：

```
$> g++ -o main foo.o source_one.o source_two.o main.o
$> ./main
x:1
x:2
```

这种方式的缺点在于，外部项目无法实例化模板，只能使用 `Foo<int>` 这一种显式实例化的模板实例。当然，倘若你正好想要禁止用户实例化模板，只允许他们使用内置的某几种实例，又要避免编译开销，这个缺点反而能够利用起来。

第二种，模板定义依旧放在头文件中，但是在其中添加显式实例化声明禁止编译器实例化模板，再手动使用显式实例化定义强制编译器生成想要的模板实例。

具体来说，将代码变为：

```
// foo.h
#ifndef FOO_H_
#define FOO_H_

#include <iostream>

template<class T>
class Foo {
public:
    Foo(T x) : data_(x) {}

    void print() const {
        std::cout << "x:" << data_ << "\n";
    }

```

```
private:
    T data_;
};

// Explicit template declaration
extern template class Foo<int>;

#endif // FOO_H_
```

然后，在 source_one.cpp 显式实例化模板，source_two.cpp 中什么都不做。代码变为：

```
// source_one.cpp
#include "foo.h"

template class Foo<int>;

void do_something1() {
    Foo<int> foo(1);
    foo.print();
}

// source_two.cpp
#include "foo.h"

void do_something2() {
    Foo<int> foo(2);
    foo.print();
}
```

于是，当前只有 source_one.cpp 中实际生成了模板实例，source_two.cpp 中没有模板实例。查看目标文件符号表，内容为：

```
// source_one.o
0000000000000000 000000000000001b W Foo<int>::Foo(int)
0000000000000000 000000000000001b W Foo<int>::Foo(int)
0000000000000000 n Foo<int>::Foo(int)
```

```
0000000000000000 0000000000000048 W Foo<int>::print() const

// source_two.o
                 U Foo<int>::Foo(int)
                 U Foo<int>::print() const
```

可见事实的确如此。若是现在链接这两个目标文件，其中也只会存在一份实例（但是没有剔除操作，本来就只存在一份模板实例）：

```
$> ld -r -o source.o source_one.o source_two.o
$> nm -C source.o
                 U _GLOBAL_OFFSET_TABLE_
000000000000009c t _GLOBAL__sub_I__Z13do_something1v
0000000000000151 t _GLOBAL__sub_I__Z13do_something2v
0000000000000000 T do_something1()
00000000000000b5 T do_something2()
000000000000004f t
↪  __static_initialization_and_destruction_0(int, int)
0000000000000104 t
↪  __static_initialization_and_destruction_0(int, int)
0000000000000000 W Foo<int>::Foo(int)
0000000000000000 W Foo<int>::Foo(int)
0000000000000000 n Foo<int>::Foo(int)
0000000000000000 W Foo<int>::print() const
                 U std::ostream::operator<<(int)
                 U std::ios_base::Init::Init()
                 U std::ios_base::Init::~Init()
                 U std::cout
0000000000000000 r std::piecewise_construct
0000000000000006 r std::piecewise_construct
0000000000000000 b std::__ioinit
0000000000000001 b std::__ioinit
```

```
U std::basic_ostream<char,
 ↪  std::char_traits<char> >&
 ↪  std::operator<< <std::char_traits<char>
 ↪  >(std::basic_ostream<char,
 ↪  std::char_traits<char> >&, char const*)
U __cxa_atexit
U __dso_handle
U __stack_chk_fail
```

接着，再将这两个文件和 `main.cpp` 链接，输出可执行文件，看程序是否能够正常运行：

```
$> g++ -o main source_two.o source_one.o main.o
$> ./main
x:1
x:2
```

一切正常，这种方式的确有效，但是也有一些缺点，将所有模板定义在一个单独的头文件当中，该文件稍有动改，依赖的所有文件都得重新编译。再者，这种 header-only 的方式，迫使用户自己实例化模板，将 header-only 去除，把实例化放到 CPP 源文件中可以避免该问题。此外，模板类型若是自定义类型，则会强制用户包含包含该类型，无法通过前置声明解决，头文件依赖性增加。

第三种，模板定义依旧放在头文件中，每个包含者增加显式实例化声明禁止隐式实例化模板。

此时，模板头文件就是本节最开始的原始实现，无需改变，在 `foo.cpp` 中写入显式实例化定义。于是，代码为：

```
1 // foo.cpp
2 #include "foo.h"
3
4 template class Foo<int>;
```

再在每个包含模板头文件的源文件中，使用显式实例化声明禁止编译器生成模板实例。实现变为：

```
1 // source_one.cpp
2 #include "foo.h"
```

```

// Explicit template declaration
extern template class Foo<int>;

void do_something1() {
    Foo<int> foo(1);
    foo.print();
}

// source_two.cpp
#include "foo.h"

// Explicit template declaration
extern template class Foo<int>;

void do_something2() {
    Foo<int> foo(2);
    foo.print();
}
```

编译生成目标文件，再次查看符号表，内容为：

```
// source_one.o
                U Foo<int>::Foo(int)
                U Foo<int>::print() const
// source_two.o
                U Foo<int>::Foo(int)
                U Foo<int>::print() const
```

可见，这两个翻译单元没有实例化模板。

最后，链接所有文件，查看结果是否正常：

```
$> g++ -o main foo.o source_one.o source_two.o main.o
$> ./main
x:1
x:2
```

结果依旧不存在问题。这种方式的缺点很明显，每个使用者都需要在源文件中使用显式实例化声明以禁止编译器生成模板实例，而他们可能会在不经意间忘记。

本节至此结束，关于隐式实例化，就是模板参数推导和模板参数替换这个流程，老生常谈，见 *The Book of Modern C++*/p63[4]。

5.5 Template Parameters

本节介绍模板参数，属于模板的输入，也就是抽象的数据类型。

5.5.1 Categories of Template Parameters

模板参数类型可以分成如下六种：

- Type
- NonType
- Type template
- Variable template
- Concept
- Universal

用代码表示为：

```
template<
  typename Type,                          // Type
  auto NonType,                           // Non-type
  template<template auto...> class Temp, // Type template
  template<template auto...> auto Var,   // Variable template
  template<template auto...> concept Concept, // Concept
  template auto Universal                     // Universal
>
```

当然，目前 C++ 标准尚不支持全部的六种类型，本节只看前三种。

[4] https://github.com/lkimuk/the-book-of-modern-cpp

5.5.2 Type Parameters

这种模板参数的类型必须是一个 type-id，平时模板代码中使用的就是此类参数。例如：

```
// Example from ISO C++
template<class T> class X {};
template<class T> void f(T t) {}

struct {} unnamed_obj;

void f() {
    struct A {};
    enum { e1 };
    typedef struct {} B;
    B b;
    X<A> x1;         // OK
    X<A*> x2;        // OK
    X<B> x3;         // OK
    f(e1);           // OK
    f(unnamed_obj); // OK
    f(b);            // OK
}
```

若是模板存在类型参数和非类型参数的重载，type-id 与表达式产生将产生歧义，此时参数类型解析为 type-id，即类型参数。如：

```
// Example from ISO C++
template<class T> void f();
template<int I> void f();

void g() {
    f<int()>(); // int() is a type-id: call the first f()
}
```

5.5.3 Non-Type Parameters

这种模板参数一般称为 NTTP(Non-Type Template Parameter)。C++20 之前，NTTP 被限制为：

- lvalue reference type (to object or to function);
- an integral type;
- a pointer type (to object or to function);
- a pointer to member type (to member object or to member function);
- an enumeration type;
- `std::nullptr_t` (since c++11)

C++20 之后，NTTP 进一步扩展为：

- an floating-point type;
- a literal class type with the following properties:
  - all base classes and non-static data members are public and non-mutable and
  - the types of all bases classes and non-static data members are structural types or (possibly multi-dimensional) array thereof.

from `https://en.cppreference.com/w/cpp/language/template_parameters`

NTTP 作为模板参数，会转换成模板参数类型的常量表达式。NTTP 类型不是抽象类型，而是具体类型的常量表达式，所以称为非类型模板参数。

如果 NTTP 是引用或指针，引用或指向的值不能是：

- a temporary object;
- a string literal object;
- the result of a `typeid` expression
- a predefined `__func__` variable;
- a subobject of one of the above.

from ISO/IEC C++ 2020 13.4.2/4

例如：

```
// Example from ISO C++
template<const int* pci> struct X {};
int ai[10];
X<ai> xi; // array to pointer and qualification conversions

struct Y {};
template<const Y& b> struct Z {};
Y y;
Z<y> z; // no conversion, but note extra cv-qualification

template<int (&pa)[5]> struct W {};
int b[5];
W<b> w; // no conversion

void f(char);
void f(int);

template<void (*pf)(int)> struct A {};

A<&f> a; // selects f(int)

template<auto n> struct B {};
B<5> b1;        // OK, template parameter type is int
B<'a'> b2;      // OK, template parameter type is char
B<2.5> b3;      // OK since C++20, template parameter type is
↪  double
B<void(0)> b4; // error, template parameter type cannot be
↪  void
```

再来看一个临时对象的例子：

```
// Example from ISO C++
template<const int& CRI> struct B {};

// error: temporary would be required for template argument
B<1> b1;
```

```

int c = 1;
B<c> b2; // OK

struct X { int n; };
struct Y { const int &r; };
template<Y y> struct C {};
// error: subobject of temporary object used to initialize
// reference member of template parameter
C<Y{ X{1}.n }> c;
```

最后，需要注意，string-literal 不能直接用作 NTTP，只可以间接通过常量指针或作为类的构造函数参数来使用：

```
// Example from ISO C++
template<class T, T p> class X {};

// error: string literal object as template-argument
X<const char*, "Studebaker"> x;
// error: subobject of string literal object as
// template-argument
X<const char*, "Knope" + 1> x2;

const char p[] = "Vivisectionist";
X<const char*, p> y; // OK

struct A {
    constexpr A(const char*) {}
};

// OK, string-literal is a constructor argument to A
X<A, "Pyrophoricity"> z;
```

5.5.4 Type Template Parameters

这是模板 Type 参数的复杂形式，也就是 Template Template Arguments。

此种情况下，模板实参只能是类模板或别名模板。当模板实参是类模板时，形参匹配只会考虑主模板，偏特化模板仅在基于此模板模板形参实例化时才会被考虑。例如：

```
/// Example from ISO C++
// primary template
template<class T> class A {
    int x;
};

// partial specialization
template<class T> class A<T*> {
    long x;
};

template<template<class U> class V> class C {
    V<int> y;
    V<int*> z;
};

// V<int> within C<A> uses the primary template,
// so c.y.x has type int;
// V<int*> within C<A> uses the partial specialization,
// so c.z.x has type long.
C<A> c;
```

模板实参和模板模板形参匹配时，后者至少要与前者同等特殊。简单来说，就是形参 P 不能比实参 A 更加抽象。这是 C++17 才修复的一个问题，之前 P 与 A 必须完全匹配，导致一些合理的实参无法匹配。比如：

```
template<template <typename> class> void FD();
template<typename, typename = int> struct SD { /* ... */ };
FD<SD>();  // OK; error before C++17
```

C++17 之后，只要满足同等特殊的规则，P 与 A 就能够匹配，如：

```
template<template <int> class> void FI();
template<template <auto> class> void FA();
```

```
template<auto> struct SA {};
template<int> struct SI {};
FI<SA>();  // OK; error before C++17
FA<SI>();  // error
```

对于 `FI<SA>()`，模板模板形参 `P` 是 `int`，而模板实参 `A` 是 `auto`，前者比后者特殊，这叫至少同等特殊。前者的特殊性只要小于等于后者，就叫同等特殊，否则便匹配失败，例如 `FA<SI>()`。

下面提供更多例子：

```
template<class T> class A {};
template<class T, class U = T> class B {};
template<class... Types> class C {};
template<auto n> class D {};

template<template<class> class P> class X {};
template<template<class...> class Q> class Y {};
template<template<int> class R> class Z {};

X<A> xa; // OK
X<B> xb; // OK since C++17
X<C> xc; // OK since C++17
Y<A> ya; // OK
Y<B> yb; // OK
Y<C> yc; // OK
Z<D> zd; // OK since C++17
```

再来看一个特殊的例子[5]：

```
#include <variant>
#include <iostream>

template<typename T>
struct TypeChecker {
    void operator()() {
```

[5] from https://stackoverflow.com/questions/72911910/exact-rules-for-matching-variadic-template-template-parameters-in-partial-templa

```
        std::cout << "I am other type\n";
    }
};

template<typename ... Ts, template<typename> typename V>
    requires std::same_as<V<Ts...>, std::variant<Ts...>>
struct TypeChecker<V<Ts...>> {
    void operator()() {
        std::cout << "I am std::variant\n";
    }
};

int main() {
    TypeChecker<std::variant<int, float>>{}();
    TypeChecker<std::variant<float>>{}();
    TypeChecker<int>{}();
}
```

这个例子中模板模板形参和模板实参是否满足至少同等特殊原则？`P` 是 `template<typename> typename V`，`A` 是 `std::variant<Ts...>`，看着无法完全匹配，实际上 `P` 与 `A` 满足至少同等特殊原则。因此输出应该为：

```
I am std::variant
I am std::variant
I am other type
```

然而，只有 GCC 和 MSVC 默认会有以上结果，Clang 的输出都是 `I am other type`。只因其默认没有应用 C++17 这个修复的匹配规则，加上 `-frelaxed-template-template-args` 编译标志，则可得到正常输出。此外，在 C++17 以前，GCC 可以添加 `-fnew-ttp-matching` 标志来得到新的匹配结果。

同时也需注意，实参并非一定要是例子中的 `std::variant`，其他支持可变模板参数的类同样适用，如 `std::tuple`，或是自定义的类型 `template<class...> struct S`。

最后，再来看两个例子，结束本节。

例子一：

```
template<class T> struct eval;

```

```
template<
    template<class, class...>class TT,
    class T1, class... Rest>
struct eval<TT<T1, Rest...>> {};

template<class T1> struct A;
template<class T1, class T2> struct B;
template<int N> struct C;
template<class T1, int N> struct D;
template<class T1, class T2, int N = 17> struct E;


// OK: matches partial specialization of eval
eval<A<int>> eA;
// OK: matches partial specialization of eval
eval<B<int, float>> eB;
// error: C does not match TT in partial specialization
eval<C<17>> eC;
// error: D does not match TT in partial specialization
eval<D<int, 17> eD;
// error: E does not match TT in partial specialization
eval<E<int, float>> eE;
```

模板形参和实参不匹配，实参中的非类型参数比模板形参更特殊，于是出错。

例子二：

```
template<typename T>
concept C = requires (T t) { t.f(); };
template<typename T>
concept D = C<T> && requires (T t) { t.g(); };

template<template<C> class P> struct S {};

template<C> struct X {};
template<D> struct Y {};
```

```
template<typename T> struct Z {};

S<X> s1; // OK, X and P have equivalent constraints
S<Y> s2; // error: P is not at least as specialized as Y
S<Z> s3; // OK, P is at least as specialized as Z
```

当有 Concept 约束时，至少同等特殊原则更易识别。若是模板模板形参和模板实参的约束一致，则约束等价；若模板模板形参比模板实参的约束多一些，则不满足至少同等特殊原则；若模板模板形参比模板实参的约束少一些，则满足。

5.6 Implementing make_index_sequence with Various Template Techniques

本节以实现 `make_index_sequence` 作为问题，展示诸多经典的模板技术。

需要注意，此处并未参考 `std::make_index_sequence` 的实现，标准实现一般考虑得更加周到，本处只是籍此需求演示各种模板技术，作为独立的例子理解即可。另外，模板和宏的代码生成机制不同，模板无法直接生成原生的索引，例如上一章中 `GMP_MAKE_INDEX_SEQUENCE(2)` 可以生成 `0, 1`，而模板参数需要依附于变量、函数或类，标准里面这个类命名为 `index_sequence`，于是对应的生成结果便为 `index_sequence<0, 1>`。

在实现 `make_index_sequence` 之前，先定义 `index_sequence`。代码为：

```
template<size_t... Is> struct index_sequence {};

template<size_t... Is>
void print(index_sequence<Is...>) {
    (std::cout << Is << " ", ...);
    std::cout << "\n";
}

int main() {
    // Output: 0 1 2
    print(index_sequence<0, 1, 2>{});
}
```

此处只是利用可变模板参数，无须絮烦。但要注意，`make_index_sequence` 实现方式不同，`index_sequence` 的实现可能也需要微调，增加一些抽象。后续小节，有差异时会明确写出，无差异时即为此默认实现。

5.6.1 Implementing make_index_sequence with Template Specialization

第一种方式采用模板偏特化，作为最经典的模板特性，暴露核心用法。

完整实现为：

```
template<size_t N, bool, size_t...>
struct make_index_sequence_helper {};

template<size_t N, size_t... Is>
struct make_index_sequence_helper<N, false, Is...> {
    using type = index_sequence<Is...>;
};

template<size_t N, size_t... Is>
struct make_index_sequence_helper<N, true, Is...> {
    using type = typename make_index_sequence_helper<
        N - 1, 0 < N - 1, Is..., sizeof...(Is)>::type;
};

template<size_t N>
using make_index_sequence = typename
    make_index_sequence_helper<N, 0 < N>::type;
```

这个实现引出了一个关键的模板技术，如何动态产生可变模板参数？

一般来说，可变模板参数由用户传入，而 `make_index_sequence<N>` 本身就是避免用户传入可变的参数，取而代之的是一个编译期常量。怎样从无到有产生可变模板参数，是一个核心技术，直接关系到编译期迭代技术。

当前的模板不支持自下而上迭代，只允许自上而下递归。因此，递归是应对动态产生式问题的根本思维。递归存在两个关键点，一是结束条件，二是链接关系。逻辑被拆成两个分支，结束为 `false`，未结束为 `true`，这正是需要使用模板偏特化的原因。结束条件很简单，生成最终的 `index_sequence`，主要逻辑处理在链接关系这个分支。链接关系逻辑分支中，每次 `N` 减一的同时，可变模

板参数增加一，怎么增加？便需要借助 `sizeof...(Is)`，它可以计算可变模板参数数量，这个数量正好与索引值对应。初始情况下，不存在可变模板参数，于是 `sizeof...(Is)` 产生 `0`，每递归一次，可变模板参数加一，直至 `N` 等于 0，步入结束条件。

测试代码为：

```
int main() {
    // Output: 0 1 2 3 4
    print(make_index_sequence<5>{});
}
```

5.6.2 Implementing make_index_sequence with constexpr if

第二种方式采用 C++17 `constexpr if`，两个分支不需再借助偏特化模板，极大降低代码重复性，流程近似动态条件分支，代码更加简洁、易于理解。

```
template<size_t N, size_t... Is>
constexpr auto make_index_sequence() {
    if constexpr (sizeof...(Is) < N)
        return make_index_sequence<
            N, Is..., sizeof...(Is)>();
    else
        return index_sequence<Is...>{};
}
```

解法思路，表过不题。且说两种方式的区别，模板偏特化采用的是类模板，而 `constexpr if` 采用的是函数模板，因此一个返回的是类型，一个返回的是编译期常量值。

5.6.3 Implementing make_index_sequence with Recursive Lambda

第三种方式采用 C++23 Recursive Lambda，无非是第二种方式的另一种表现形式。

```
constexpr auto make_index_sequence =
    []<size_t N, size_t... Is>(this auto self) {
    if constexpr (sizeof...(Is) < N)
        return self.template operator()<
```

```
            N, Is..., sizeof...(Is)>();
    else
        return index_sequence<Is...>{};
};

print(make_index_sequence.template operator()<5>());
```

Template Lambda 是 C++20 加入的特性，调用的其实是 `template<size_t N, size_t... Is> operator()`，所以必须使用这种奇怪的语法来传递参数。

若想避免这种语法，可以为 Lambda 增加一个 `int_constant` 参数，用法变为：

```
template<size_t...> struct int_constant {};

constexpr auto make_index_sequence =
    []<size_t N, size_t... Is>
    (this auto self, int_constant<N, Is...>) {
        if constexpr (sizeof...(Is) < N)
            return self(int_constant<N, Is...,
                sizeof...(Is)>{});
        else
            return index_sequence<Is...>{};
};

print(make_index_sequence(int_constant<5>{}));
```

较于前者，此法略微简化了一些用法，但仍麻烦。更加精妙的一种方式是组合泛型 Lambda 和变量模板，添上类型能力。于是，就可以像前面两种实现方式那般使用：

```
template<size_t N>
inline constexpr auto make_index_sequence =
    []<size_t... Is>(this auto self) {
    if constexpr (sizeof...(Is) < N)
        return self.template operator()<
            Is..., sizeof...(Is)>();
    else
```

```
        return index_sequence<Is...>{};
};

print(make_index_sequence<5>());
```

这种方式和第二种方式 `constexpr if` 同源，返回的依旧是一个编译期常量值。

5.6.4 Implementing make_index_sequence with Tag Dispatching

第四种方式采用标签分发，本质是利用标签区分函数重载，选择不同的分支，是早期的一种模板技术。

完整实现如下：

```
template<size_t N, size_t... Is>
constexpr auto make_index_sequence_impl(std::false_type) {
    return index_sequence<Is...>{};
}

template<size_t N, size_t... Is>
constexpr auto make_index_sequence_impl(std::true_type) {
    return make_index_sequence_impl<N, Is..., sizeof...(Is)>(
        std::bool_constant<sizeof...(Is) + 1 < N>{});
}

template<size_t N>
constexpr auto make_index_sequence() {
    return make_index_sequence_impl<N>(
        std::bool_constant<0 < N>{});
}
```

这里直接借助标准中的已有标签 `std::false_type` 和 `std::true_type` 实现分发，分别对应两个逻辑分支，实现递归。`std::bool_constant` 能够生成这两个标签，从而完成实际分发。

此法与模板偏特化相似，但本质上属于不同的多态表现方式。

5.6.5 A Faster Implementation of make_index_sequence

本节介绍一种 $O(\log N)$ 复杂度的实现[6]，编译速度更快，但相应地，也更复杂。

完整实现为：

```
template<size_t... Is> struct index_sequence {
    using type = index_sequence;
};

template<class, class> struct merge_index_sequence {};

template<size_t... LIs, size_t... RIs>
struct merge_index_sequence<
    index_sequence<LIs...>, index_sequence<RIs...>>
    : index_sequence<LIs..., (sizeof...(LIs) + RIs)...>
{};

template<size_t N>
struct make_index_sequence
    : merge_index_sequence<
        typename make_index_sequence<N / 2>::type,
        typename make_index_sequence<N - N / 2>::type>
{};

template<>
struct make_index_sequence<0> : index_sequence<> {};
template<>
struct make_index_sequence<1> : index_sequence<0> {};
```

这是一种与递归继承相似的实现思路，前面的实现思路都是线性递增，而这种思路每次会折半实例化，直接把实现化深度从 $O(N)$ 降低到 $O(\log N)$。核心思路并无变化，依旧是递归，利用 `sizeof...(Is)` 逐次增加模板参数。变化主要在于折半实例化与合并，问题被拆分得更小。

以 `N` 取 5 为例，此时被拆分成求 2 和 3，代码表示为：

[6]borrowed from `https://stackoverflow.com/a/17426611/19868918`

```
merge_index_sequence<
    make_index_sequence<2>, make_index_sequence<3>>
↑
make_index_sequence<5>
```

这是第一层深度，问题转换成求 `make_index_sequence<2>` 和 `make_index_sequence<3>`。

先看 `make_index_sequence<2>`，又被拆分成求 1 和 2，代码表示为：

```
merge_index_sequence<
    make_index_sequence<1>, make_index_sequence<0>>
↑
make_index_sequence<2>
```

到此时就不能再拆分，否则 `1 - 1/2` 将造成死循环，以是 `make_index_sequence<1>` 和 `make_index_sequence<0>` 特化作为结束条件，`make_index_sequence<2>` 最终生成：

```
index_sequence<0, 1>
↑
make_index_sequence<2>
```

取 `type`，即为 `index_sequence<0, 1>`。

再看 `make_index_sequence<3>`，被拆分成 1 和 2，代码表示为：

```
merge_index_sequence<
    make_index_sequence<1>, make_index_sequence<2>>
↑
make_index_sequence<3>
```

同理可得，最终生成 `index_sequence<0, 1, 2>`。

以上是第二层深度，问题拆分便已解决，至此合并结束。利用 `merge_index_sequence` 合并 `index_sequence<0, 1>` 和 `index_sequence<0, 1, 2>`，最终得到 `index_sequence<0, 1, 2, 3, 4>`，问题解决。

注意，前文提到这种思路与递归继承相似，但其并非递归继承，只是递归实例化的一种形式，远无递归继承复杂。

5.7 Type Traits

类型萃取就是从已有模板类型中提取信息的方法，是模板的核心技术之一。

以 std::function 为例，如何萃取模板参数的基本信息？整体思路是增加间接层，将想要的参数重新写到模板头。

实现一个 function_traits 萃取信息，代码如下：

```
template<typename T>
struct function_traits;

template<typename R, typename... Args>
struct function_traits<std::function<R(Args...)>> {
    static constexpr std::size_t size = sizeof...(Args);
    using result_type = R;

    template<size_t I>
    struct get {
        using type = typename std::tuple_element<
            I, std::tuple<Args...>>::type;
    };
};
```

这个工具能够萃取 std::function 的参数个数、返回类型及指定位参数类型。使用示例：

```
template<class F, size_t I>
void print_ith_type() {
    std::cout << typeid(typename
        function_traits<F>::get<I>::type).name() << "\n";
};

int main() {
    using FuncType =
        std::function<void(int, double, std::string)>;

    std::cout << "size of template parameter list: "
```

```
        << function_traits<FuncType>::size << "\n";
    std::cout << typeid(function_traits<FuncType>
        ::result_type).name() << "\n";

    print_ith_type<FuncType, 0>();
    print_ith_type<FuncType, 1>();
    print_ith_type<FuncType, 2>();
}
```

当然可以进一步简化完善，此处只展示这种方式，不细论。

但也有某些特殊的，以 `is_member_function_pointer` 的一种实现方式为例：

```
template<class T>
struct is_member_function_pointer_helper
    : std::false_type {};

template<class T, class U>
struct is_member_function_pointer_helper<T U::*>
    : std::is_function<T> {};

template<class T>
struct is_member_function_pointer
    : is_member_function_pointer_helper<
        std::remove_cv_t<T> > {};
```

注意这里的 `<T U::*>`，`T` 和 `U::*` 分别对应哪部分类型？倘若成员函数为 `void A::foo()`，此时 `T` 对应 `void()`，`U::*` 对应 `A::foo`。

在模板元编程中，类型萃取并不算难，不再多论。

5.8 Summary

本章深入回顾了模板涉及的核心理论和技术，是产生式模板元编程的基本知识。

模板所带来的泛型思维、模板参数、模板类型、模板实例化、类型萃取是其中的核心，再高深的模板技术，也以此为基石。某些概念讨论得鞭辟入里，比较深刻，即便是模板书籍中亦未涉及，有一定难度，但对于产生式元编程来说，很是必要。

后续章节，将利用这些技术实现代码生成，编写强大的工具。

Chapter 6

Advanced Template Techniques

模板编程，技巧如云，妙艺似雨。第五章滔滔滚滚地讲述了模板的核心概念和常用技术，篇幅稍长，诸般妙诀，未遑悉论，放于本章。

闲话不题，本章尽是一些巧妙的实践，于极尽模板产生式编程大有用处，难度不低，重要重要！

本章代码非是示例，而是 GMP[1] 库的真实代码。

6.1 Type List

Type List 是个只装类型的容器，不含任何数据、函数或类型成员，定义相当简单。代码如下：

```
template<typename...> struct type_list {};
```

可变模板参数是实现的核心，表示 Type List 中存入的类型。类型并无一致性要求，是以 Type List 本身还是个异构容器。为何需要这种容器？只因编译期可能需要对很多类型操作，借助这种容器，能够方便地保存各种各样的类型。

可是就一行代码，也没见能进行什么操作呀？其实，Type List 只是个类型容器，需要配套算法才能发挥作用，这些算法才是关键。算法可以分成如下几类：

- 容积（Capacity）
  - 大小（`type_list_size_v`）

[1] `https://github.com/lkimuk/gmp`

  - 判空（type_list_empty）

- 元素访问（Element access）
  - 索引访问（type_list_element_t）
  - 前向类型（type_list_front）
  - 后向类型（type_list_back）
  - 尾部类型（type_list_tail）
  - 内含检测（type_list_contains_v）
- 元素操作（Element manipulation）
  - 合并（type_list_concat_t）
  - 移除（type_list_remove_t）
  - 前向移除（type_list_pop_front）
  - 后向移除（type_list_pop_back）
  - 插入（type_list_insert_t）
  - 前向插入（type_list_push_front）
  - 后向插入（type_list_push_back）
  - 反转（type_list_reverse_t）
  - 去重（type_list_unique_t）
  - 过滤（type_list_filter_t）

搭配以上这些基础算法，Type List 方得显现妙用。以下各个小节，便来展示各个算法的实现，这些实现并不一定是最优的，但却能展示模板技巧。GMP 后续会进一步优化各个算法。

Note

GMP 命名上，以 _v 结尾表示实现为变量模板，该变量模板借助非 _v 版本实现，如 type_list_size_v，其实是 type_list_size::value 的简写。以 _t 结尾表示实现为别名模板，该模板是非 _t 版本的简化别名，如 type_list_element_t，其实是 type_list_element::type 的简写。无特殊标识结尾，表示实现直接借助了 _v 或 _t 版本的已有组件，如 type_list_empty，其实是 type_list_size_v 在大小是否为 0 时的简写，不命名为 type_list_empty_v，是因为没有相对应的 type_list_empty::value 写法。

6.1.1 Capacity

容积类算法，主要包含一些与大小有关的操作，Type List 不曾分配内存，大小相关的操作实际指的是与可变模板参数数量相关的操作。

6.1.1.1 Size(type_list_size_v)

若要获得 Type List 的大小，只需借助 `sizeof...()` 计算可变模板参数的个数，并非难事。

```
/// class type_list_size
template<typename> struct type_list_size;

template<typename... Types>
struct type_list_size<type_list<Types...>>
    : std::integral_constant<std::size_t, sizeof...(Types)>
    {};

template<typename T>
inline constexpr std::size_t type_list_size_v =
    type_list_size<T>::value;
```

在实际开发中，为将代码重复降至最低，一般会借助 `std::integral_constant` 工具类，简化代码。

6.1.1.2 Empty(type_list_empty)

判空可以直接利用 `type_list_size_v`，于是实现进一步简化为：

```
template<type_list_like T>
inline constexpr bool type_list_empty =
    (type_list_size_v<T> == 0);
```

`type_list_like` 是检测类型是否为 Type List 的 Concept，目前组件不全，无法实现。等几个关键的算法实现完成之后，再来展示其实现。

6.1.2 Element Access

元素访问类算法，主要包含类型访问的一些操作，即类型的获取操作。

6.1.2.1 Access Elements(type_list_element_t)

索引式访问是元素访问的根本实现，只要能够按索引访问，再想实现前向访问和后向访问便是水到渠成的事。

完整实现，如下所示：

```
/// class type_list_element
template<std::size_t Idx, typename T>
requires (Idx < type_list_size_v<T>)
struct type_list_element;

template<std::size_t Idx, typename Head, typename... Types>
struct type_list_element<Idx, type_list<Head, Types...>>
    : std::type_identity<typename
        type_list_element<Idx-1, type_list<Types...>>::type>
{};

template<typename Head, typename... Types>
struct type_list_element<0, type_list<Head, Types...>>
    : std::type_identity<Head>
{};

template<std::size_t Idx, typename T>
using type_list_element_t = type_list_element<Idx, T>::type;
```

索引访问的核心思路，就是依次遍历，返回指定索引的类型。但模板不支持自下而上迭代，只支持自上而下递归，因此遍历与常规运行期编程差异甚大。每次递归都需要舍弃头类型，再以尾类型作为新的 Type List，直到递归终止，此时头类型就是欲访问的类型。

`std::type_identity` 是 C++20 加入的一个辅助模板编程的类型，用于提升抽象层次，减少代码重复。其内部不过是抽象了一个 `type` 类型成员而已，这个类型成员就是传入的模板参数，以是免于重复编写。再者，GMP 库基于 C++20 编写，使用 `std::type_identity` 并无丝毫问题。

6.1.2.2 `type_list_like` Concept

本节原本不应该出现在此处，内容不属于索引访问的子节，但又不得不置于此处，因为其实现依赖于 `type_list_size_v` 和 `type_list_element_t`，置于前面无法写，后续组件又需用到，只剩这一处合适。

`type_list_like` 的作用是保证类型输入安全，倘若这些算法的类型输入不是 Type List，可能会产生未定义的行为，用户难以定位错误。需要一种方式，能够检测类型是 Type List，而 Type List 只是个带有可变模板参数的简单类型，很多用户类型都可能满足这一条件，所以检测起来并不容易。必须找出 Type List 的关键特征，区别开其他类型，才能确保类型属于 Type List。

所幸这个问题已经有人考虑过了，就是 C++23 增加的 `tuple-like` Concept，用于识别 `std::tuple`。照猫画虎，`type_list_like` 的实现为：

```
/**
 * \brief Concept to check if a type is type_list-like.
 *
 * \tparam T The type to be checked.
 */
template<typename T>
concept type_list_like
    = !std::is_reference_v<T> && requires(T t) {
        typename type_list_size<T>::type;
        requires std::derived_from<type_list_size<T>,
            std::integral_constant<
                std::size_t, type_list_size_v<T>>
        >;
    } && []<std::size_t... Is>(std::index_sequence<Is...>) {
        return (requires(T) {
            typename std::integral_constant<
                Is, std::remove_const_t<T>>;
        } && ...);
    }(std::make_index_sequence<type_list_size_v<T>>{});
```

核心涉及两个已有组件，`type_list_size` 和 `type_list_element_t`。输入 `T` 若是类 Type List 类型，第一点便要满足 `type_list_size<T>::type`，并且 `type_list_size` 继承自 `std::integral_constant`；第二点稍微复杂，需要能够正常使用

std::integral_constant 访问其所含有的所有类型，借助 std::index_sequence 实现遍历。

由此，只有满足 type_list_like 约束的类型，才能作为其后所有实现算法的输入。

6.1.2.3 Front Type(type_list_front)

前向类型就是获得 Type List 中的第一个类型，直接借助 type_list_element_t 实现即可。

```
template<type_list_like T>
using type_list_front = type_list_element_t<0, T>;
```

6.1.2.4 Back Type(type_list_back)

后向类型就是获得 Type List 中的最后一个类型，可以直接借助 type_list_element_t 和 type_list_size_v 实现。

```
template<type_list_like T>
using type_list_back =
    type_list_element_t<type_list_size_v<T> - 1, T>;
```

6.1.2.5 Tail Type(type_list_tail)

尾部类型就是获取 Type List 中的非首位类型，只要移除第一个类型，剩下的即是尾部类型，因此可以直接借助后续小节实现的 type_list_pop_front 实现。

```
template<type_list_like T>
using type_list_tail = type_list_pop_front<T>;
```

只是一个简单的类型别名而已。

6.1.2.6 Contains(type_list_contains_v)

用于检测 Type List 中是否包含指定的类型，是返回 true，否则返回 false。

实现为：

```
/// contains
template<typename, type_list_like>
```

```
struct type_list_contains;

template<typename U, typename... Types>
struct type_list_contains<U, type_list<Types...>>
    : std::bool_constant<(std::same_as<Types, U> || ...)>
{};

template<typename U, type_list_like T>
inline constexpr bool type_list_contains_v =
    type_list_contains<U, T>::value;
```

核心逻辑就是利用 Fold Expressions 逐个对比类型，只要有一个为真，则整个条件为真，后续不会执行（短路原则）。

6.1.3 Element Manipulation

元素访问主要是类型的获取操作，而元素操作则集中于元素的修改操作。修改操作会改变 Type List，或增加类型，或删除类型。

修改操作比获取操作的实现要复杂许多。

6.1.3.1 Concat(type_list_concat_t)

合并操作将多个 Type List 合并成一个 Type List，例如三个 Type List 分别为 `type_list<int>`、`type_list<double>` 和 `type_list<char, int>`，合并之后，产生一个新类型 `type_list<int, double, char, int>`。

实现思路是将每次合并前两个 Type List，将其结果再和剩余的 Type List 合并，重复这个过程。如此一来，复杂的大问题就被拆解成为一个个小问题，降低解决难度。

代码为：

```
/// class type_list_concat
template<
    type_list_like TypeList1,
    type_list_like TypeList2,
    type_list_like... RestTypeLists>
struct type_list_concat
    : std::type_identity<typename type_list_concat<typename
```

```
        type_list_concat<TypeList1, TypeList2>::type,
        RestTypeLists...>::type>
{};

template<typename... LTypes, typename... RTypes>
struct type_list_concat<type_list<LTypes...>,
    type_list<RTypes...>>
    : std::type_identity<type_list<LTypes..., RTypes...>>
{};

template<type_list_like... TypeLists>
using type_list_concat_t =
    type_list_concat<TypeLists...>::type;
```

只需专心处理合并两个 Type List 的小问题，此问题便迎刃而解。

6.1.3.2 Remove(type_list_remove_t)

移除操作将删除 Type List 中指定位置的类型，返回移除后的 Type List。对于运行期编程来说，这种操作并不困难，而此处是编译期的类型移除，只能使用模板，颇为不易。

首先，定义好元类型形式，确定输入类型和安全约束。

```
/// class type_list_remove
template<std::size_t Idx, type_list_like T>
    requires (Idx < type_list_size_v<T>)
struct type_list_remove;
```

type_list_remove 接受两个模板参数，第一个是索引值，第二个是一个 Type List，索引值必须小于其长度。

其次，确定思路，理清递归逻辑。每次递归，索引减一，取出类型列表中的头类型，将其保存到头类型列表。随着递归执行，头类型列表中的类型会依次增加，类型列表中的类型会依次减少。当递归终止，舍弃将放入头类型列表中的新类型，合并头类型列表和类型列表，便是移除之后的新 Type List。

这个逻辑需要额外的模板参数，以表示头类型列表，于是需要一个新的辅助模板 type_list_remove_impl。

先来看 type_list_remove 的偏特化定义，核心逻辑转发至 type_list_remove_impl 实现。代码如下：

```
/// class type_list_remove
template<std::size_t Idx, typename Head, typename... Types>
struct type_list_remove<Idx, type_list<Head, Types...>>
    : std::type_identity<typename
        detail::type_list_remove_impl<Idx-1,
            type_list<Types...>, type_list<Head>>::type>
{};

template<typename Head, typename... Types>
struct type_list_remove<0, type_list<Head, Types...>>
    : std::type_identity<type_list<Types...>>
{};
```

当索引为 0 时，则需单独处理，否则数据将溢出，遂单独特化。

再来看 `type_list_remove_impl` 的具体实现：

```
/// remove impl
template<std::size_t, type_list_like, type_list_like>
struct type_list_remove_impl;

template<std::size_t Idx, typename Head,
    typename... Types, typename... Heads>
struct type_list_remove_impl<Idx,
    type_list<Head, Types...>, type_list<Heads...>>
    : std::type_identity<typename
        type_list_remove_impl<Idx-1, type_list<Types...>,
            type_list<Heads..., Head>>::type>
{};

template<typename Head,
    typename... Types, typename... Heads>
struct type_list_remove_impl<0,
    type_list<Head, Types...>, type_list<Heads...>>
    : std::type_identity<type_list_concat_t<
        type_list<Heads...>, type_list<Types...>>>
{};
```

逻辑思路前文已交代清楚，不再絮烦。到递归终止时，舍弃此时的头类型，再使用 type_list_concat_t 合并头类型列表和随着递归不断减少的类型列表，即是移除类型之后的 Type List。

最后，别忘了设置简化别名。

```
template<std::size_t Idx, type_list_like T>
using type_list_remove_t = type_list_remove<Idx, T>::type;
```

type_list_remove_impl 为内部类，用户不可使用，type_list_remove 用来有些麻烦，让用户直接使用 type_list_remove_t，以简化代码。

基于已有实现，前向移除和后向移除的实现不过是小菜一碟。代码如下：

```
// removes the first type
template<type_list_like T>
using type_list_pop_front = type_list_remove_t<0, T>;

// removes the last type
template<type_list_like T>
using type_list_pop_back =
    type_list_remove_t<type_list_size_v<T> - 1, T>;
```

6.1.3.3 Insert(type_list_insert_t)

插入操作，在 Type List 的指定位置插入新的类型。

整体实现和移除操作的逻辑大体相同，分成三步走。

第一步，定义元类型形式，确定输入类型及其安全性。

```
template<std::size_t Idx, typename, type_list_like T>
    requires (Idx <= type_list_size_v<T>)
struct type_list_insert;
```

与移除操作不同，插入操作的索引位置可以和 Type List 的大小相等，表示在列表的末尾插入类型。

第二步，确定思路。依旧是依次取出头类型，放入头类型列表，类型列表每次只余尾类型，递归终止时，合并头类型列表、新类型和尾类型即可。当在 Type List 的开头插入类型时，单独处理，直接创建一个新的 Type List。

当然，实际参数依旧复杂，需要借助 type_list_insert_impl 辅助类。typ」e_list_insert 本身只作简单处理，实际逻辑由辅助类完成。

type_list_insert 代码如下：

```cpp
template<std::size_t Idx, typename NewType,
    typename Head, typename... Types>
    requires (Idx > 0)
struct type_list_insert<Idx, NewType,
    type_list<Head, Types...>>
    : std::type_identity<typename
        detail::type_list_insert_impl<Idx-1, NewType,
            type_list<Head>, type_list<Types...>>::type>
{};

template<typename NewType, typename... Types>
struct type_list_insert<0, NewType, type_list<Types...>>
    : std::type_identity<type_list<NewType, Types...>>
{};
```

`type_list_insert_impl` 是实现的核心所在，代码为：

```cpp
/// insert impl
template<std::size_t, typename, type_list_like,
↪  type_list_like>
struct type_list_insert_impl;

template<std::size_t Idx, typename NewType, typename Head,
    typename... Heads, typename... Types>
    requires (Idx > 0)
struct type_list_insert_impl<Idx, NewType,
    type_list<Heads...>, type_list<Head, Types...>>
    : std::type_identity<typename type_list_insert_impl<
        Idx-1, NewType, type_list<Heads..., Head>,
        type_list<Types...>>::type>
{};

template<typename NewType,
    typename... Heads, typename... Types>
struct type_list_insert_impl<0, NewType,
    type_list<Heads...>, type_list<Types...>>
```

```
    : std::type_identity<type_list<Heads...,
        NewType, Types...>>
{};
```

没甚稀奇，不另细述。

功能完成之后，再为其添加简化使用的别名模板。

```
template<std::size_t Idx, typename NewType, type_list_like T>
using type_list_insert_t = type_list_insert<Idx, NewType,
↪  T>::type;
```

基于此，不难实现前向插入和后向插入。代码为：

```
// inserts a type to the beginning
template<typename NewType, type_list_like T>
using type_list_push_front = type_list_insert_t<0, NewType,
↪  T>;

// adds a type to the end
template<typename NewType, type_list_like T>
using type_list_push_back =
↪  type_list_insert_t<type_list_size_v<T>, NewType, T>;
```

6.1.3.4 Reverse(type_list_reverse_t)

反转操作，就是逆序现有的 Type List 类型。比如，`type_list<int, double, char>` 反转后为 `type_list<char, double, int>`。

原本实现这个功能比较复杂，但借助前面已经实现的组件，复杂度显著降低。

整体的实现思路，就是每次利用 `type_list_back` 取出 Type List 的后向类型，装入一个新的 Type List，再利用 `type_list_pop_back` 移除输入 Type List 的后向类型。循环往复，直到递归结束，所有类型已全归新位。

基本实现为：

```
/// reverse
template<type_list_like> struct type_list_reverse;

template<typename... Types>
```

```
struct type_list_reverse<type_list<Types...>>
    : std::type_identity<typename
        detail::type_list_reverse_impl<
            type_list_size_v<type_list<Types...>> - 1,
            type_list<type_list_back<type_list<Types...>>>,
            type_list_pop_back<type_list<Types...>>>::type>
{};

template<>
struct type_list_reverse<type_list<>>
    : std::type_identity<type_list<>>
{};

template<type_list_like T>
using type_list_reverse_t = type_list_reverse<T>::type;
```

具体实现需要借助 `type_list_reverse_impl` 辅助类，它的实现如下：

```
/// reverse impl
template<std::size_t, type_list_like, type_list_like>
struct type_list_reverse_impl;

template<std::size_t Idx,
    typename... NewTypes, typename... Types>
struct type_list_reverse_impl<Idx,
    type_list<NewTypes...>, type_list<Types...>>
    : std::type_identity<
        typename type_list_reverse_impl<Idx-1,
            type_list<NewTypes...,
                type_list_back<type_list<Types...>>>,
            type_list_pop_back<type_list<Types...>>
        >::type>
{};

template<typename... NewTypes>
struct type_list_reverse_impl<0,
```

```
    type_list<NewTypes...>, type_list<>>
    : std::type_identity<type_list<NewTypes...>>
{};
```

思路便是前面的思路，递归终止时，新的 Type List 长度将和输入 Type List 的长度相等，输入 Type List 的长度将变为 0。

6.1.3.5 Unique(type_list_unique_t)

去重操作，用于移除 Type List 中的重复类型，例如，`type_list<int, double, int, double, double>` 去重后为 `type_list<int, double>`。

这个操作比之前的所有操作都要复杂，因为它要既存在针对当前类型的操作，也存在针对过去类型的操作。每个当前对比的类型，都要和之前的类型对比，以保证不存在重复。但是，我们已经实现过内含检测操作（`type_list_contains_v`），它可以直接检测是否重复，只要返回为真，则重复，否则为未重复。于是，将前面用到的思路全部整合起来，每次取出头类型，检测其是否出现在头类型列表，若未存在，则加入头类型列表，否则遗弃。同时，Type List 类型也在每轮递归中依次减少头类型，以保证递归正常执行。

`type_list_unique` 的实现如下：

```
/// unique
template<type_list_like> struct type_list_unique;

template<typename Head, typename... Types>
struct type_list_unique<type_list<Head, Types...>>
    : std::type_identity<typename
        detail::type_list_unique_impl<
            type_list_size_v<type_list<Head, Types...>> - 1,
            type_list<>, type_list<Types...>, Head, false>
        ::type>
{};

template<>
struct type_list_unique<type_list<>>
    : std::type_identity<type_list<>>
{};

```

```
template<typename T>
using type_list_unique_t = type_list_unique<T>::type;
```

初始化时，头类型列表为空，故为 type_list<>，其中必然不存在当前的头类型，初始条件直接置为 false 即可。

实际工作被转移至 type_list_unique_impl 辅助类，由于需要接收条件判断的结果，如今实现都需要变成双份，一份处理 true 的情况，一份处理 false 的情况。代码为：

```
/// unique impl
template<std::size_t, type_list_like,
    type_list_like, typename, bool>
struct type_list_unique_impl;

template<std::size_t Idx,
    typename Head, typename CompType,
    typename... NewTypes, typename... Types>
struct type_list_unique_impl<Idx, type_list<NewTypes...>,
    type_list<Head, Types...>, CompType, true>
    : std::type_identity<typename type_list_unique_impl<
        Idx-1, type_list<NewTypes...>, type_list<Types...>,
        Head, type_list_contains_v<
            Head, type_list<NewTypes...>>>::type>
{};

template<std::size_t Idx,
    typename Head, typename CompType,
    typename... NewTypes, typename... Types>
struct type_list_unique_impl<Idx, type_list<NewTypes...>,
    type_list<Head, Types...>, CompType, false>
    : std::type_identity<typename type_list_unique_impl<
        Idx-1, type_list<NewTypes..., CompType>,
        type_list<Types...>, Head, type_list_contains_v<
            Head, type_list<NewTypes..., CompType>>>::type>
{};

```

```
template<typename CompType,
    typename... NewTypes, typename... Types>
struct type_list_unique_impl<0, type_list<NewTypes...>,
    type_list<Types...>, CompType, true>
    : std::type_identity<type_list<NewTypes..., Types...>>
{};

template<typename CompType,
    typename... NewTypes, typename... Types>
struct type_list_unique_impl<0, type_list<NewTypes...>,
    type_list<Types...>, CompType, false>
    : std::type_identity<
        type_list<NewTypes..., CompType, Types...>>
{};
```

只要明确以上逻辑思路，这个代码理解起来并不复杂，不敢饶舌。

6.1.3.6 Filter(type_list_filter_t)

过滤操作，指的是根据特定条件，移除满足条件的所有元素。

如第五章所言，C++ 当前只支持三种类型的模板参数，Variable template、Concept 和 Universal 这三种类型尚不支持，因此表示条件只能借助类模板作为模板参数来实现。

声明如下：

```
/// filter
template<type_list_like, template<typename> class Pred>
    requires
    std::same_as<std::remove_const_t<decltype(Pred<void>::value)>,
    bool>
struct type_list_filter;

template<type_list_like T, template<typename> class Pred>
using type_list_filter_t = type_list_filter<T, Pred>::type;
```

其中，`Pred` 表示条件，需要使用模板模板参数表示，可自行定义，但是这个类型必须提供一个 `bool` 型的 `value` 成员，作为条件。例如：

```cpp
template<typename T>
using IntType = std::bool_constant<std::same_as<T, int>>;

// generates `type_list<int, int>`
type_list_filter_t<type_list<int, int, double>, IntType>
```

将 `IntType` 作为 `Pred` 传入，将过滤掉所有非 `int` 类型，只留下 `int` 类型。

实现思路和去重操作类似，只是加入头类型列表的判断条件由 `Pred` 提供，不再固定。只有满足条件，才将类型加入头类型列表，其他类型将被舍弃。

先来看 `type_list_filter` 的基本操作，和 `type_list_unique` 相似，代码为：

```cpp
template<
    template<typename> class Pred,
    typename Head, typename... Types>
struct type_list_filter<type_list<Head, Types...>, Pred>
    : std::type_identity<
        typename detail::type_list_filter_impl<
            type_list_size_v<type_list<Head, Types...>> - 1,
            type_list<>, type_list<Types...>, Pred, Head,
            Pred<Head>::value>::type>
{};

template<template<typename> class Pred>
struct type_list_filter<type_list<>, Pred>
    : std::type_identity<type_list<>>
{};
```

再来看辅助类 `type_list_filter_impl` 的核心实现，依旧与 `type_list_unique_impl` 相似，代码如下：

```cpp
template<
    std::size_t,
    type_list_like,
    type_list_like,
    template<typename> class,
    typename, bool>
struct type_list_filter_impl;
```

```
8
9  template<std::size_t Idx,
10     template<typename> class Pred,
11     typename Head, typename CompType,
12     typename... Heads, typename... Types>
13 struct type_list_filter_impl<Idx, type_list<Heads...>,
14     type_list<Head, Types...>, Pred, CompType, true>
15     : std::type_identity<typename type_list_filter_impl<
16         Idx-1, type_list<Heads..., CompType>,
17         type_list<Types...>, Pred, Head,
18         Pred<Head>::value>::type>
19 {};
20
21 template<std::size_t Idx,
22     template<typename> class Pred,
23     typename Head, typename CompType,
24     typename... Heads, typename... Types>
25 struct type_list_filter_impl<Idx, type_list<Heads...>,
26     type_list<Head, Types...>, Pred, CompType, false>
27     : std::type_identity<typename type_list_filter_impl<
28         Idx-1, type_list<Heads...>, type_list<Types...>,
29         Pred, Head, Pred<Head>::value>::type>
30 {};
31
32 template<
33     template<typename> class Pred,
34     typename CompType,
35     typename... Heads, typename... Types>
36 struct type_list_filter_impl<0, type_list<Heads...>,
37     type_list<Types...>, Pred, CompType, true>
38     : std::type_identity<type_list<Heads...,
39         CompType, Types...>>
40 {};
41
42 template<
```

```
43      template<typename> class Pred,
44      typename CompType,
45      typename... Heads, typename... Types>
46  struct type_list_filter_impl<0, type_list<Heads...>,
47      type_list<Types...>, Pred, CompType, false>
48      : std::type_identity<type_list<Heads..., Types...>>
49  {};
```

虽说实现甚为复杂，模板参数亦是不少，但有前面各种其他操作的实现过渡，循序渐进，由浅及深，已不难理解，故不再赘述。

6.2 Compile-Time Loop

屡有写到，遍历方式分为两种，自下而上叫迭代，自上而下叫递归。

模板存在多种类型，包含变量模板、函数模板和类模板，使用最多的便是递归这种遍历方式，因为目前只有递归才是函数和类共同支持的机制。若是需要操纵类型，像 Type List 的诸多算法，就只能使用递归。函数模板和变量模板都没有操纵类型的能力，它们直接返回的是一个值，而非类型，尽管可以借助 `decltype(val)` 从值推导出类型，却会绕个圈子，语法上也不直观，所以类模板是唯一可以使用的方式。这意味着，`if constexpr` 这种简化递归的方式无法使用，它们需要依赖函数使用，而类似 Type List 这种情况，不存在一个函数。

Type List 在使用时不用生成变量，只需要类型本身，但也有很多情境需要的是值，如 `std::tuple`，此时就多了一种遍历——迭代。相较于递归，迭代有诸多优势，一是易于理解，二是支持随机访问，三是无须额外的终止函数或特化类。递归需要保存很多中间类型，往往需要借助辅助类才能完成略显复杂的工作，而迭代只存在于一个函数里面，不必增加模板参数来保存中间变量。

总而言之，迭代是一种更加自然的遍历方式，本节便来介绍此类技巧。

6.2.1 Expansion Statements(C++26 maybe)

Expansion Statements 就是 *template for*，编译期的循环语法，过去几年写的反射文章里面已进过多次，但今年该提案终于被人重拾，更发布了 P1306R3[2]，本节重写一下本特性，更新一下内容。

Expansion Statements 能够直接生成一些重复性的语句，而不必使用复杂的的模板实例化机制。例如，借助这种方式迭代 `std::tuple`，可以这样写：

[2] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p1306r3.pdf

```
auto tup = std::make_tuple(0, 'a', 3.14);
template for (auto elem : tup)
    std::cout << elem << std::endl;
```

template for 语句中的代码，将对 `std::tuple` 的每个元素展开，生成如下代码：

```
auto tup = std::make_tuple(0, 'a', 3.14);

{
    auto elem = std::get<0>(tup);
    std::cout << elem << std::endl;
}

{
    auto elem = std::get<1>(tup);
    std::cout << elem << std::endl;
}

{
    auto elem = std::get<2>(tup);
    std::cout << elem << std::endl;
}
```

这种方式直接了当，语法形式和运行期的 `for` 循环并无太大差别，用来更加方便。但是需要区分概念，Expansion Statements 并不是循环，而是对提供的语句进行一系列的实例化展开，以达到循环的效果。变量在每次实例化都会重新展开，类型可以不同，而真正的循环不会对每个迭代元素都生成一份代码，类型是相同的。因此，它的名字叫 Expansion Statements，而不是 Compile-time for。

在新的修订版中，**expansion-init-list** 语法更加简洁，允许直接像下面这样写：

```
template for (auto elem : {0, 'a', 3.14})
    std::cout << elem << std::endl;
```

`{0, 'a', 3.14}` 中的大括号只是标记展开的元素范围，并非初始化列表。在 R3 版本中，展开过程中还可以执行 `break` 和 `continue` 操作。如：

```
// Prints: 1 3 5
template for (auto v : { 1,2,3,4,5,6,7,8,9 }) {
    if (v % 2 == 0) continue;

    std::print("v: {} ", v);
    if (v % 5 == 0) break;
}
```

在语法表现上，这种方式已经和 `for` 语句非常相近，纵使是第一次看到这样的代码，也能够猜出它的作用。

Expansion Statements 允许展开的内容如下：

- 可析构的类（包含原始结构体和 `std::tuple`）
- Constexpr ranges（包含编译期 `std::vector`）
- 大括号分割的表达式列表（**expansion-init-list**，包含参数包展开）

此外，为了方便计数实例化展开次数，一个 `enumerate` 辅助功能也被添加了进来，举个例子：

```
template for (auto x : enumerate(some_tuple)) {
    // x has a count and a value
    std::println("{}: {}", x.count, x.value);

    // The count is also a compile-time constant.
    using T = decltype(x);
    std::array<int, T::count> a;
}
```

这个辅助函数不难实现，只是简单地返回了一个元素为计数和值的元组适配器。

总体来说，Expansion Statements 通常和静态反射配套使用更有用，过去已经写过数篇文章，此处不再烦絮。

6.2.2 Compile-time for

Expansion Statements 对应于 Range-based for 语句，只能集中处理局部元素，还有另一种原始的 for 语句，存在索引下标，能够从整体视角下处理元素，意即

不仅可以操作当前元素，还可以操作任何其他位置的元素。对于某些复杂的情况，我们需要这种能够随机访问任意位置的编译期循环方式。

举个例子，Type List 反转算法的实现，除了当前的思路，还有另一种思路，就是利用 Type List 长度减去每次迭代的索引值，得到的值就是反转之后的类型索引，再用 `type_list_element_t` 直接取得对应类型，存入一个新的 Type List 中便可了事。此处将以 `std::tuple` 作为示例，思路完全一致，不同之处在于 `std::tuple` 存在数据成员，一般是作为变量使用，所以不必局限于类模板，可以使用函数模板。

反转 `std::tuple` 的实现代码如下：

```
void reverse_of(auto... args) {
    auto tuple = std::make_tuple(
        std::forward<decltype(args)>(args)...);
    return [&tuple]<auto... Is>(std::index_sequence<Is...>) {
        return std::make_tuple(
            std::get<sizeof...(args) - 1 - Is>(tuple)...);
    }(std::index_sequence_for<decltype(args)...>{});
}
```

这里暴露了频繁被使用的一个迭代技巧——Compile-Time for。索引通过 `std::index_sequence` 表示，不像运行期迭代时那样需要初始索引、终止条件和索引递增/递减，编译期索引直接包含所有索引值，可同时表示运行期迭代的这三个语句，再结合 Fold Expressions 进行实际迭代。

将语法抽象出来，便可知 Compile-Time for 的基本表达形式为：

```
[]<auto... I>(std::index_sequence<I...>) {
    // write your code
}(std::make_index_sequence<N>{})
```

这个迭代方式很有用，比如用它来实现一个遍历 `std::tuple` 的函数，代码如下：

```
template<typename Tuple>
void for_each(const Tuple& tuple, auto f) {
    [&tuple, &f]<auto... I>(std::index_sequence<I...>) {
        (f(std::get<I>(tuple)), ...);
    }(std::make_index_sequence<std::tuple_size_v<Tuple>>{});
```

```
}


int main() {
    auto tuple = std::make_tuple(0, 'a', 3.14);

    // 0 a 3.14
    for_each(tuple, [](auto val) {
        std::cout << val << " ";
    });
}
```

基于该技巧，灵活运用，能够实现许多强大的工具，虽然语法有些奇怪，但这其实就是目前的编译期循环语法。GMP 也会大量使用该技巧，实现不计其数的元编程组件。

6.3 Recursive Inheritance

若论模板当中最复杂最精妙的技术是什么，那无疑是递归继承。该技术源自 Andrei Alexandrescu 二十多年前的著作 *Modern C++ Design*，随书产生的 Loki 库将模板技术运用到了极致。

递归继承就是根据可变模板参数，递归地继承类型，如上一章中展示的经典 `std::tuple` 实现法。此节所说的递归继承更深一层，指的是泛化的递归继承，无具体类型，复杂度飙升。

具体来说，递归继承又可分成线性递归继承和散乱递归继承。

6.3.1 Linear Recursive Inheritance

线性递归继承，继承形状犹如一串糖葫芦，从底部依次往上继承。`std::tuple` 的实现便是线性递归继承最简单的形式，每次继承自身，传递尾部模板参数，直到 `std::tuple<>`。本节谈论的情境更加复杂，指的是泛化的线性递归继承，即允许定制的线性递归继承。

泛化是由特殊到一般的过程，是对具体情境抽象化后的结果。若 `std::tuple` 这般，以线性递归继承逐次往类型当中注入成员变量，只是众多产生式需求中的一种，某些需求可能想要往类型当中注入成员函数、虚函数等其他代码，

不论是何种需求，都是注入代码。因此，提取注入定制点是泛化的第一步。这个提取的注入点称为 MetaFun，表示生成代码的元函数，函数签名如下：

```
template<typename T, typename Root>
class MetaFun {};
```

T 表示当前的具体产品，Root 是包含剩余产品的继承体系。例如，tuple<int, double> 若采用这种泛化实现，将先继承自 MetaFun<int, tuple<double>>，再继承自 tuple<double>，次再继承 MetaFun<double, tuple<>>，最后继承 tuple<>。可见，泛化的线性递归继承是个交替继承的过程，每次通过 MetaFun 注入代码，通过自身解参递归继承。线性递归继承需要包含结束条件，表示最后继承的 Root，某些类型可能没有特定的继承，泛化实现提供了一个空类型表示默认的结束条件，定义如下：

```
/// Empty class
struct empty_type {};
```

结束条件既定，定制点和继承流程亦明，于是泛化实现可以实现为：

```
namespace gmp
{

/// Generates linear hierarchy
template<typename T,
    template<typename, typename> class MetaFun,
    typename Root = empty_type>
struct gen_linear_hierarchy : MetaFun<T, Root>
{};

template<typename... Types,
    template<typename, typename> class MetaFun,
    typename Root>
struct gen_linear_hierarchy<
    type_list<Types...>, MetaFun, Root>
    : MetaFun<type_list_front<type_list<Types...>>,
        gen_linear_hierarchy<type_list_tail<
            type_list<Types...>>, MetaFun, Root>>
```

```
{};

template<typename T,
    template<typename, typename> class MetaFun,
    typename Root>
requires std::derived_from<MetaFun<T, Root>, Root>
struct gen_linear_hierarchy<type_list<T>, MetaFun, Root>
    : MetaFun<T, Root>
{};

} // namespace gmp
```

MetaFun 作为模板参数，而其自身也是模板，由是需要使用模板模板参数。其实，在产生式元编程中，只要表示定制点，皆需使用模板模板参数，实现的功能就类似于泛化算法实现中 Lambda 表示的定制点。

线性递归继承的泛化实现并不复杂，思路却杳冥难测，流程细微巧绝，可注入定制逻辑，产生成千上万行代码。

6.3.2 Scatter Recursive Inheritance

散乱递归继承，继承形状宛如竹栅栏，由底向上，左右开枝。

这种方式以多继承消除了线性递归继承那样交替的继承体系，左边的继承表示具体产品注入代码，右边的继承表示剩余产品继承体系。因为不依赖交替继承，所以 MetaFun 也不再需要 Root 参数，剩余产品为一则是递归的结束条件。

于是实现变为：

```
namespace gmp
{

/// Generates scatter hierarchy
template<typename T, template<typename> class MetaFun>
struct gen_scatter_hierarchy : MetaFun<T> {};

template<typename... Types, template<typename> class MetaFun>
struct gen_scatter_hierarchy<type_list<Types...>, MetaFun>
    : gen_scatter_hierarchy<
```

```
        type_list_front<type_list<Types...>>, MetaFun>,
      gen_scatter_hierarchy<
        type_list_tail<type_list<Types...>>, MetaFun>
{};

template<typename T, template<typename> class MetaFun>
struct gen_scatter_hierarchy<type_list<T>, MetaFun>
  : MetaFun<T>
{};

} // namespace gmp
```

实现依旧不甚复杂，复杂的是思路。每次将当前的具体产品，即 Type List 的前向类型送入递归终点，用元函数产生定制的代码，再将剩余产品（Type List 的尾部类型）传入自身进行递归继承，直到剩余产品为一，递归结束。

散乱递归继承可以和线性递归继承组合起来使用，将散乱递归继承作为线性递归继承的 `Root`，当线性递归继承结束之时，便是散乱递归继承开始之时。通常，线性递归继承用于产生类型的具体接口，而散乱递归继承用于产生类型的抽象接口，两相结合，便既生成了抽象接口，又生成了具体实现。下一小节包含一个具体实例。

6.3.3 Best Practice: Implement a Generic Abstract Factory

本实例源于数年前写的设计模式系列文章中的抽象工厂一章，即 okdp[3] 库的实现，GMP 库将旧的实现全部进行了更新，将设计模式作为一个单独的子模块提供泛型组件。本章里的递归继承是面向高级开发者的讲解，几年前写的抽象工厂文章[4]里面含有图例讲解递归继承细节，可与本章结合阅读，降低理解难度。

抽象工厂用于产生多系列具体产品，每个产品的创建接口和具体实现一致，遂可泛化以减少代码重复。实现分为两部分，抽象实现和具体实现，抽象实现提供接口，具体实现提供真实逻辑，前者可由散乱递归继承实现，后者可由线性递归继承实现。

首先，创建抽象接口的元函数，以生成抽象实现的相关代码。如下：

[3] https://github.com/lkimuk/okdp

[4] https://mp.weixin.qq.com/s/fF2Hd3oIGoTqMrfIdPd0MQ

```
template<typename T>
struct abstract_factory_meta_fun
{
    virtual T* do_create(std::type_identity<T>) = 0;
    virtual ~abstract_factory_meta_fun() {}
};
```

主要包含创建类型的 `do_create()` 接口，接受一个标签参数，用于在生成的众多接口中识别目标类型的准确接口。

其次，使用散乱递归继承具体实现抽象工厂的抽象部分。完整代码：

```
namespace detail
{

template
<
    type_list_like AbstractProductList,
    template<typename> class MetaFun =
↪   abstract_factory_meta_fun
>
struct abstract_factory_impl
    : gen_scatter_hierarchy<AbstractProductList, MetaFun>
{
    using ProductList = AbstractProductList;

    template<class T, typename... Args>
    requires type_list_contains_v<T, AbstractProductList>
    T* create(Args&&... args)
    {
        MetaFun<T>& meta = *this;
        return meta.do_create(std::type_identity<T>{});
    }

    template<class T, typename... Args>
    requires type_list_contains_v<T, AbstractProductList>
    std::shared_ptr<T> create_shared(Args&&... args)
```

```
    {
        return std::shared_ptr<T>(
            create<T>(std::forward<Args>(args)...));
    }

    template<class T, typename... Args>
    requires type_list_contains_v<T, AbstractProductList>
    std::unique_ptr<T> create_unique(Args&&... args)
    {
        return std::unique_ptr<T>(
            create<T>(std::forward<Args>(args)...));
    }
};

} // namespace detail

template<typename... Types>
using abstract_factory =
↪   detail::abstract_factory_impl<type_list<Types...>>;
```

每个类型都将通过散乱递归继承，以 `abstract_factory_meta_fun` 生成特定接口。创建类型之时，用户将调用 `create()` 接口，实际创建将转发调用元函数生成的 `do_create()` 抽象接口。`create_shared()` 和 `create_unique()` 是依赖 `create()` 接口创建的智能指针版本，使用起来更加方便，但需注意，欲持有已有指针，不能使用 `std::make_shared` 等工厂方法，而应使用原始的代理类型。

接着，创建具体实现的元函数，以生成实际创建对象的逻辑。如下：

```
template<typename ConcreteProduct, typename Base>
struct creation_meta_fun : Base
{
    using ProductList =
        type_list_tail<typename Base::ProductList>;

    /*template<typename... Args>*/
    ConcreteProduct* do_create(
        std::type_identity<type_list_front<
```

```
            typename Base::ProductList>>) override
    {
        return new ConcreteProduct;
    }
};
```

这个元函数主要供线性递归继承使用，故而拥有一个 Base 参数，用来交替继承。元函数中具体实现了 do_create() 接口，创建具体类型。此处的关键在于将具体实现的接口和抽象接口一一对应，每次都要通过 Type List 算法取出产品列表当中的第一个产品，以标签分发识别。ProductList 每次递归都只余尾部类型，如此递归才会终止。

最后，使用线性递归继承具体实现抽象工厂的具体部分。完整代码：

```
template
<
    typename AbstractFactory,
    type_list_like ConcreteProductList,
    template<typename, typename>
    class Creator = creation_meta_fun
>
struct concrete_factory : gen_linear_hierarchy<
    type_list_reverse_t<ConcreteProductList>,
    Creator, AbstractFactory>
{};
```

具体工厂线性递归的终点是抽象工厂散乱递归的起点，因此第一个模板参数 AbstractFactory 就是前面实现的抽象部分。第二个模板参数是具体的产品列表，第三个是生成代码的元函数，默认设置为 creation_meta_fun。

每次 Creator（即 creation_meta_fun ）取出具体产品列表中的第一个产品作为 do_create() 实现的返回类型，需要和标签对应，否则类型将不一致，导致出错。但是，creation_meta_fun 从 Base::ProductList 中取出的第一个抽象产品类型的顺序和具体产品的取出顺序正好相反，于是需要把具体产品列表 Con」creteProductList 通过 Type List 反序算法调整一下顺序，才能确保正确展开对应的类型接口。

这一整个泛化过程甚为复杂，大家要记住，问题复杂性是不能消除的，只能转移。想要为一类问题抽象出一个统一解法，问题本身就是极为复杂的，但

是这个复杂性可以转移到库的开发者身上，对用户隐藏。即便是以后标准支持更简单的做法，那也不过是将复杂性转移到了标准身上，对开发者隐藏。只要能够对用户隐藏复杂性，提供简单易用的接口，库这边的底层实现复杂点并不是问题。

下面是利用上述 GMP 实现，生成抽象工厂代码的应用例子。

```
#include <gmp/dp/abstract_factory.hpp>
#include <iostream>

struct Lux {
    virtual ~Lux() = default;
    virtual void print() = 0;
};

struct EasyLux : Lux {
    void print() override {
        std::cout << "easy level lux mode\n";
    }
};

struct HardLux : Lux {
    void print() override {
        std::cout << "hard level lux mode\n";
    }
};

struct DieLux : Lux {
    void print() override {
        std::cout << "die level lux mode\n";
    }
};

struct Teemo {
    virtual ~Teemo() = default;
    virtual void print() = 0;
};
```

```
31
32 struct EasyTeemo : Teemo {
33     void print() override {
34         std::cout << "easy level Teemo mode\n";
35     }
36 };
37
38 struct HardTeemo : Teemo {
39     void print() override {
40         std::cout << "hard level Teemo mode\n";
41     }
42 };
43
44 struct DieTeemo : Teemo {
45     void print() override {
46         std::cout << "die level Teemo mode\n";
47     }
48 };
49
50 struct Ziggs {
51     virtual ~Ziggs() = default;
52     virtual void print() = 0;
53 };
54
55 struct EasyZiggs : Ziggs {
56     void print() override {
57         std::cout << "easy level Ziggs mode\n";
58     }
59 };
60
61 struct HardZiggs : Ziggs {
62     void print() override {
63         std::cout << "hard level Ziggs mode\n";
64     }
65 };
```

```
66
67 struct DieZiggs : Ziggs {
68     void print() override {
69         std::cout << "die level Ziggs mode\n";
70     }
71 };
72
73 using AbstractAIFactory =
74     gmp::abstract_factory<Ziggs, Lux, Teemo>;
75 using EasyLevelAIFactory =
76     gmp::concrete_factory<AbstractAIFactory,
77         gmp::type_list<EasyZiggs, EasyLux, EasyTeemo>>;
78 using HardLevelAIFactory =
79     gmp::concrete_factory<AbstractAIFactory,
80         gmp::type_list<HardZiggs, HardLux, HardTeemo>>;
81 using DieLevelAIFactory =
82     gmp::concrete_factory<AbstractAIFactory,
83         gmp::type_list<DieZiggs, DieLux, DieTeemo>>;
84
85 int main() {
86     auto factory = std::make_shared<DieLevelAIFactory>();
87     auto ziggs = factory->create_unique<Ziggs>();
88     auto lux = factory->create_shared<Lux>();
89     auto teemo = factory->create_shared<Teemo>();
90
91     ziggs->print();
92     lux->print();
93     teemo->print();
94 }
```

用户无需再手动为类型重复编写抽象工厂的相关代码，直接使用 `gmp::ab」stract_factory` 和 `gmp::concrete_factory` 生成实际代码。

最终输出为：

```
die level Ziggs mode
die level lux mode
```

```
die level Teemo mode
```

6.4 Pretty Printing Type Names

打印类型名称，听来似是一个很简单的需求，但对于还没有反射的 C++ 而言，尚非易事。

类型属于 Type，对象属于 Value，前者是编译期的东西，后者则是运行期的东西。你可以打印一个变量的值，却无法打印一个类型的名称。那么如何才能实现这个需求？

通常来说，解决问题的思路是将新问题转换为已经存在解决方案的旧问题。其一，编译期目前只能输出错误信息，这个错误信息也可以是一种打印类型名称的方法。我们需要做的，就是主动触发报错，可以利用重载决议的相关知识达到这个目的。其二，既然无法直接打印类型，那么就将类型转换为 Value，从而在运行期进行打印。但是，通过表格暴力转换法其实并不可行，因为类型组合起来实在太多了。此时可以借助一些语言或编译器特性来获取到类型信息，比如通过 `typeid` 就可以根据类型得到一个简单的名称。

思路既定，接着就可以顺着这个思路设计实现，以下各节展示各种实作法。

6.4.1 Leveraging Overload Resolution

这种思路是利用错误信息输出类型信息，如何触发错误，如果大家已经读过*洞悉函数重载决议*[5]，相信已经有了深刻认识。

具体实现如下：

```
1 template<typename...> struct type_name {};
2 template<typename... Ts> struct name_of {
3     using X = typename type_name<Ts...>::name;
4 };
5
6 int main() {
7     name_of<int, float, const char*>();
8 }
```

Name Lookup 查找的名称 `name_of` 带有模板，于是会进入重载决议的第二阶段 Template Handling。模板参数已经显式指出，因此其实并不会进行 TAD，而

[5] *The Book of Modern C++* Chapter 1

是直接模板参数替换。但是，编译器发现 `type_name<Ts...>::name` 并没有 `name` 类型，于是模板替换失败，产生 Hard Error，编译失败。

这个 Hard Error 信息，就带有类型名称，输出如下：

```
error: no type named 'name' in 'struct type_name<int, float,
↪  const char*>'
7 |     using X = typename type_name<Ts...>::name;
```

现在，就可以使用该实现在编译期查看实际类型的名称，比如：

```
template<typename T>
void f(T t) {
    name_of<decltype(t)>();
}

int main() {
    const int i = 1;
    f(i);
}
```

输出为：

```
error: no type named 'name' in 'struct type_name<int>'
7 |     using X = typename type_name<Ts...>::name;
```

可以看到，TAD 在推导参数时丢弃了 Top-level `const` 修饰，`t` 的实际类型为 `int`。

这种方式的优点是，完全发生于编译期，且实现简单，缺点也很明显，无法指定输出形式，看起来不够直观。

6.4.2 Demanged Name

另一种方式是借助 `typeid` 关键字，通过它可以获得一个 `std::type_info` 对象，结构如下：

```
namespace std {
    class type_info {
    public:
        virtual ~type_info();
        bool operator==(const type_info& rhs) const noexcept;
```

```
        bool before(const type_info& rhs) const noexcept;
        size_t hash_code() const noexcept;
        const char* name() const noexcept;
        type_info(const type_info&) = delete;
        type_info& operator=(const type_info&) = delete;
    };
}
```

其中的成员函数 `name()` 就可以返回类型的名称，这样就根据 Type 获取到了 Value。但是，标准说这个名称是基于实现的：

> Returns an implementation defined null-terminated character string containing the name of the type. No guarantees are given; in particular, the returned string can be identical for several types and change between invocations of the same program.

事实也的确如此，MSVC 返回的是一段可读的类型名称，而 GCC 和 Clang 返回的是 Mangled Name。如调用 `typeid(int).name()`，GCC 和 Clang 返回的是 `i`，而 MSVC 返回的是 `int`。

但幸好，它们内部提供了 Demangle API，通过相关 API 便可将类型名称转换为可读的名称。这个 API 定义如下：

```
namespace abi {
extern "C" char* __cxa_demangle (const char* mangled_name,
                    char* buf,
                    size_t* n,
                    int* status);
} // namespace abi
```

主要关注第一个参数即可，其他参数皆可置空。第一个参数就是 `type_info::name()` 返回的 Mangled Name，返回值为 Demangled Name。因此，可以分而论之，MSVC 直接使用 `type_info::name()` 返回的类型名称；对于 GCC/Clang，则先使用 Demangle API 进行解析，次再使用。

具体实现如下[6]：

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

[6] borrowed from `https://stackoverflow.com/a/20170989`

```cpp
#include <typeinfo>
#include <type_traits>
#ifndef _MSC_VER
    #include <cxxabi.h>
#endif // _MSC_VER

template<typename T>
std::string type_name() {
    using type = typename std::remove_reference<T>::type;

    // 1. 通过 typeid 获得类型名称
    const char* name = typeid(type).name();
    std::string result;

    // 2. 通过 GCC/Clang 扩展 API 获得 Demangled Name
#ifndef _MSC_VER
    char* demangled_name =
        abi::__cxa_demangle(name, nullptr, nullptr, nullptr);
    result += demangled_name;
    free(demangled_name);
#else
    result += name;
#endif // _MSC_VER

    // 3. 添加丢弃的修饰
    if (std::is_const<type>::value)
        result += " const";
    if (std::is_volatile<type>::value)
        result += " volatile";
    if (std::is_lvalue_reference<T>::value)
        result += "&";
    if (std::is_rvalue_reference<T>::value)
        result += "&&";

    return result;
```

```
38 }
39
40 struct Base {};
41 struct Derived : Base {};
42
43 int main() {
44     std::cout << type_name<const int&>() << "\n";
45     std::cout << type_name<Base>() << "\n";
46     std::cout << type_name<Derived>() << "\n";
47 }
```

实现分为三个步骤，注释已经写得足够清晰，这里补充两个重点：

1. Demangled API 的返回值是采用 `malloc()` 分配的内存，需要手动进行释放；
2. `type_info::name()` 会丢弃 CV 及引用修饰符，所以需要手动再添加这些修饰。

最终各个编译器的输出内容如下：

```
MSVC:
int const&
struct Base
struct Derived

GCC/Clang:
int const&
Base
Derived
```

这种方式的优点在于，可以统一格式，输出清晰，缺点在于，实现稍微麻烦，要考虑更多情况，且发生于运行期。

6.4.3 Compiler Extensions

编译器还存在另一种扩展，含有类型信息。大家也许用过 `__func__`，这是每个函数内部都会预定义的一个标识符，表示当前函数的名称。于 C99 添加到 C 标准，C++11 添加到 C++ 标准，定义如下：

```
static const char __func__[] = "function-name";
```

C++ 引入的这个是 “implementation-defined string”，表明也是基于实现的，不过在三个平台上的输出基本是一致的。这个标识符只包含函数名称，并不会附带模板参数信息。但是与其相关的扩展附带有这部分信息，GCC/Clang 的扩展为 `__PRETTY_FUNCTION__`，MSVC 的扩展为 `__FUNCSIG__`。它们的内容形式也是基于实现的，一个简单的例子：

```
template<typename T>
consteval auto type_name() {
#ifdef _MSC_VER
    return __FUNCSIG__;
#elif defined(__GNUC__)
    return __PRETTY_FUNCTION__;
#elif defined(__clang__)
    return __PRETTY_FUNCTION__;
#endif
}

int main() {
    std::cout << type_name<int>();
}
```

输出分别为：

```
// GCC
consteval auto type_name() [with T = int]

// Clang
auto type_name() [T = int]

// MSVC
auto __cdecl type_name<int>(void)
```

GCC 的格式不错，Clang 丢弃了 `consteval`，MSVC 同样如此，但它加上了函数调用约定。

现在需要做的，就是根据这些信息，解析出想要的信息。可以借助 `std::`」

`string_view` 在编译期完成这个工作。具体实现如下[7]：

```
template<typename T>
consteval auto type_name() {
    std::string_view name, prefix, suffix;
#ifdef __clang__
    name = __PRETTY_FUNCTION__;
    prefix = "auto type_name() [T = ";
    suffix = "]";
#elif defined(__GNUC__)
    name = __PRETTY_FUNCTION__;
    prefix = "consteval auto type_name() [with T = ";
    suffix = "]";
#elif defined(_MSC_VER)
    name = __FUNCSIG__;
    prefix = "auto __cdecl type_name<";
    suffix = ">(void)";
#endif
    name.remove_prefix(prefix.size());
    name.remove_suffix(suffix.size());

    return name;
}
```

通过使用 `std::string_view`，以上代码全都发生于编译期。

这种实现方式比 Demanged Name 方式好，不会丢失修饰，类型信息完善，且发生于编译期。缺点也有，编译器扩展一般都是基于实现的，没有标准保证，内容形式可能会改变，依赖于此的实现并不具备较强的稳定性。

6.5 Friend Injection

Friend Injection（友元注入）是一种巧妙的模板元编程技术，能够为模板特化提供动态行为。

这项技术充分利用了 `friend` 函数的一些特性。

一是 `hidden friend` 只能被 ADL 找到：

[7] borrowed from `https://stackoverflow.com/a/56766138`

```cpp
template<int N>
struct Tag {
    // hidden friend
    // can only be found through ADL
    friend constexpr auto flag(Tag) -> void {
        std::cout << "test\n";
    }
};


int main() {
    Tag<0> tag;
    flag(tag); // ok
}
```

这个我在 *The Book of Modern C++* Chapter 1 已经详尽讲解，不再细述。

二是 `flag()` 是否定义取决于 `Tag` 是否实例化：

```cpp
template<int N>
struct Tag {
    friend constexpr auto flag() -> void {
        std::cout << "test\n";
    }
};

constexpr auto flag() -> void;

flag(); // error, flag is undefined
```

使用之前，要么显式实例化，要么隐式实例化，才能够动态为 `flag()` 提供定义：

```cpp
constexpr auto flag() -> void;

// explicit template instantiation
template struct Tag<0>;

flag(); // ok
```

而显式模板实例化不会对名称进行访问检查，利用这个特性，甚至可以在外部访问私有数据成员，如：

```
class S {
    int data; // data is private
public:
    void print() {
        std::cout << data << "\n";
    }
};

template<int S::* Member>
struct Tag {
    friend constexpr auto flag(S& s) -> int& {
        return s.*Member;
    }
};

constexpr auto flag(S& s) -> int&;

// The usual access checking rules do not apply to
// names used to specify explicit instantiations.
template struct Tag<&S::data>;

int main() {
    S s;
    flag(s) = 42;
    s.print(); // 42
}
```

`S::data` 是私有数据成员，直接将其地址通过显式模板参数实例化，于是通过 Friend Injection 突破了数据访问防线。当然，若是使用隐式模板实例化，是无法绕过这种检查的。

三是以 `if constexpr` 配合以上两点特性，动态控制 `flag()` 的定义：

```
template<int N>
struct Tag {
```

```
    friend auto flag(Tag);
};

template<int N>
struct Writer {
    friend auto flag(Tag<N>) {}
    int value = N;
};

template<int N = 0>
constexpr void f() {
    if constexpr (requires { flag(Tag<N>{}); }) {
        std::cout << "defined flag\n";
    } else {
        std::cout << "undefined flag\n";
        Writer<N> writer;
    }

    if constexpr (requires { flag(Tag<N>{}); }) {
        std::cout << "defined flag\n";
    } else {
        std::cout << "undefined flag\n";
    }
}

int main() {
    // Output:
    // undefined flag
    // defined flag
    f();
}
```

第一次调用 `if constexpr` 检查时，`flag(Tag<0>)` 并没有定义，所以输出 “undefined flag”；之后，调用 `Writer<N>` 对其进行了隐式实例化，它实例化了，内部的 `flag(Tag<0>)` 也就跟前实例化，于是有了定义，第二次再调用检查时，自然就输出 “defined flag” 了。

以上便是该技术的核心原理。

6.6 Compile-Time Ticket Counter

Friend Injection 的经典应用场景是实现一个编译期计数器，下面展示了这种用法：

```
template <std::size_t N> struct reader {
    friend auto flag(reader);
};

template <std::size_t N> struct writer {
    friend auto flag(reader<N>) {}
    std::size_t value = N;
};

template <auto N = 0, auto = []{}>
consteval std::size_t next() noexcept {
    if constexpr (requires { flag(reader<N>{}); })
        return next<N + 1>();
    else
        return writer<N>{}.value;
}

static_assert(next() == 0);
static_assert(next() == 1);
static_assert(next() == 2);
static_assert(next() == 3);
static_assert(next() == 4);
static_assert(next() == 5);
```

这个示例充分将上述 Friend Injection 技术和多种模板元技巧相结合，实现了动态实例化类、在编译期递增计数器的效果。

`reader` 中的 `friend flag()` 函数只有声明，定义是在 `writer` 中提供的，所以在没有实例化 `writer` 之前，`flag()` 是未定义的。注意 `flag()` 的参数是模板类，每一个 `writer<N>` 都会定义一个与模板参数相关的重载。

`next()` 中则充分利用 `if constexpr` 和 `requires` 约束来在编译期判断 `flag<reader<N>>` 是否已经存在定义，是则递归调用，增加计数，否则调用 `writer<N>`，这会产生隐式实例化，进而定义 `flag<reader<N>>`。

`next()` 的模板参数 `auto = []{}` 是借助 Lambda 类型的唯一性来保证其本身能够正常递归，这是 C++20 才允许的一个小特性。

6.7 Compile-Time Get the Number of Struct Members

获取结构体成员个数是反射众多功能中的一粒沙，2016 年之后，出现了 T1 级别的实现手法，无需再像 T0 级实现方式那样得手动注册或侵入。

该功能最终会产生一个值，并且依赖遍历，由是如今最简洁的实现手法是借助函数模板和 `if constexpr`。经典的实现代码如下所示：

```
struct Any { template <class T> operator T() const; };

template<class T, class... Args>
requires std::is_aggregate_v<T>
consteval auto CountAggregateMembers() {
    if constexpr (requires { T{ Args{}... }; }) {
        return CountAggregateMembers<T, Args..., Any>();
    } else {
        return sizeof...(Args) - 1;
    }
}

struct S {
    int a;
    double b;
    std::string c;
};

int main() {
    // 3
    std::cout << CountAggregateMembers<S>();
}
```

核心思路不算复杂，就是构造一个能够隐式转换为任何类型的对象逐次去初始化目标类型，起初先用一个 `Any` 去初始化 `S` 对象，之后依次增加，一旦增加的个数超过目标类型的成员个数，初始化便会失败，此即为递归终止条件。终止时，当前的 `Any` 个数刚好比目标类型的成员个数多一，减去便是所求的结构体成员个数。

这个实现简单，是因为既用到了 C++17 `if constexpr`，又用到了 C++20 Concepts，前者极大简化了传统模板递归方式，后者极大简化了传统模板约束方式。

C++17 之前的遍历方式也可以用来解决同样的问题，但相对来说要麻烦一些。下面列举几种其他方式，改写上述代码。

第一种，借助 SFINAE，递归以传统方式实现。

```
template <class T, class... Args>
std::enable_if_t<std::is_aggregate_v<T> &&
↪   !std::is_constructible_v<T, Args...>, std::size_t>
CountAggregateMembers() {
    return sizeof...(Args) - 1;
}

template <class T, class... Args>
std::enable_if_t<std::is_aggregate_v<T> &&
↪   std::is_constructible_v<T, Args...>, std::size_t>
CountAggregateMembers() {
    return CountAggregateMembers<T, Args..., Any>();
}
```

传统方式实现函数模板递归，不可避免地需要多写一个函数来表示结束条件，是以表达同样的逻辑，得多写一些重复的代码。SFINAE 就更别提了，本来就是偶然发现的模板约束技巧，易用性和可读性自然不比专门为模板约束而设计的 Concepts。

还有一点需要注意，`std::is_constructible_v` 在 C++20 之前并不支持聚合类型的构造检测，必须为类型提供构造函数。像下面这样使用将存在问题：

```
struct S {
    int a;
    double b;
    std::string c;
};
```

```

// Compile time error before C++20
static_assert(std::is_constructible_v<S, Any, Any, Any>);
```

因此，上述 SFINAE 实现存在隐患，为解决这一问题，可以自己实现一个 `is_aggregate_constructible` 替换 `std::is_constructible_v`。代码如下：

```
template<typename T, typename... Args>
class is_aggregate_constructible {
private:
    template<typename U, typename... A>
    static auto test(int) ->
    ↪ decltype(U{std::declval<A>()...}, std::true_type{});

    template<typename, typename...>
    static auto test(...) -> std::false_type;

public:
    static constexpr bool value =
        decltype(test<T, Args...>(0))::value;
};

template<typename T, typename... Args>
inline constexpr bool is_aggregate_constructible_v =
    is_aggregate_constructible<T, Args...>::value;
```

利用这个工具，便可保证检测无误。

```
// OK
static_assert(is_aggregate_constructible_v<S, Any, Any,
↪ Any>);
```

第二种，借助 Compile-time for，以迭代替换递归完成遍历。

```
template<class T, std::size_t N>
concept ConstructibleWithN = requires {
    []<std::size_t... I>(std::index_sequence<I...>)
        -> decltype(T{ (I, Any{})... }) {
```

```
        return {};
    }(std::make_index_sequence<N>{});
};

template <class T, std::size_t N>
concept CanAggregate = std::is_aggregate_v<T> &&
    ConstructibleWithN<T, N> && !ConstructibleWithN<T, N +
    1>;

template <class T>
constexpr auto CountAggregateMembers =
    []<std::size_t... I>(std::index_sequence<I...>) {
    std::size_t R;
    ((CanAggregate<T, I> ? (R = I, true) : false) || ...);
    return R;
}(std::make_index_sequence<64>{});
```

逻辑思路不变，只是改变了遍历方式。实现变得更加复杂，主要是因为 Compile-Time for 的索引在编译期就已生成，而当前问题并无法明确索引范围，只能预先提供一个相对较大的索引范围，并且要借助 Fold Expressions 的一些高级技巧来展开逻辑。仅是提供一种思路，此问题的解决方式其实更适合自上而下的递归。

第三种，Tag Dispatching，这种方式也是利用函数重载。

```
template<typename T, typename... Args>
std::size_t CountAggregateMembersImpl(std::false_type) {
    return sizeof...(Args) - 1;
}

template<typename T, typename... Args>
std::size_t CountAggregateMembersImpl(std::true_type) {
    return CountAggregateMembersImpl<T, Args..., Any>(
        std::integral_constant<bool, std::is_aggregate_v<T>
    &&
            is_aggregate_constructible_v<T, Args..., Any>>{}
    );
```

```
}

template<typename T>
std::size_t CountAggregateMembers() {
    return CountAggregateMembersImpl<T>(std::true_type{});
}
```

逻辑与第一种方式完全一致，只是借助标签分发不同逻辑而已。看看即可，已不推荐此种方式。

6.8 Compile-Time Get the Names of Struct Members

获取结构体成员名称是反射中的另一个小小元函数，以前只有 T0 级的实现方式，需要手动注册成员信息。2023 年 12 月，有人分享了一种 T1 级的实现技巧，不几日，reflect-cpp 等反射库皆用新的方式替换了 T0 级手法。

简化的实现，代码如下：

```
template<class T> extern const T external;

template<auto Ptr>
consteval auto name_of() -> std::string_view {
    const auto name = std::string_view{
        std::source_location::current().function_name() };
#if defined(__clang__)
    const auto split = name.substr(0, name.find("}]"));
    return split.substr(split.find_last_of(".") + 1);
#elif defined(__GNUC__)
    const auto split = name.substr(0, name.find(");"));
    return split.substr(split.find_last_of(":") + 1);
#elif defined(_MSC_VER)
    const auto split =
        name.substr(0, name.find_last_of("}"));
    return split.substr(split.find_last_of(">") + 1);
#endif
}

```

```
template<std::size_t Idx, typename T>
consteval auto get(T&& t) {
    auto&& [a, b] = t;

    if constexpr (Idx == 0) return &a;
    else return &b;
}

template<std::size_t Idx, typename T>
consteval auto member_name() -> std::string_view {
    constexpr auto name = name_of<get<Idx>(external<T>)>();
    return name;
}
```

借助 `member_name`，可以获取指定位置的结构体成员名称。示例：

```
struct S {
    int foo;
    char bar;
};

int main() {
    // Output: bar
    std::cout << gmp::member_name<1, S>();
}
```

该实现技巧的核心并不出奇，就是 §6.4 讲到的思路，就是通过编译器扩展结合 `std::string_view` 手动解析类型名称。奇妙之处，在于如何触发出包含成员名称的信息？此法巧妙地结合了 `magic_get` 的实现技巧——利用 Structure Bindings 获取指定位置的结构体成员值，再通过 NTTP 将获取到的结构体成员地址作为指针传入函数模板，从而使函数名称当中包含成员变量的名称信息。此外，另一个技巧是借助 `extern` 声明一个外部变量模板，这个场景只需在编译期使用 `external<T>` 产生类型信息，并不在运行期使用该变量，而外部变量无需定义，可以避免实例化产生过多的模板变量。

需要注意，示例中的实现并不完善，`get` 尚只支持固定数量的结构体，但扩展起来并非难事，只需借助上一节编译期获取结构体成员个数所实现的工具，假设工具名为 `member_count`，那么实现可以像下面这样扩展：

```cpp
template<std::size_t Idx, typename T>
consteval auto get(T&& t)
{
    if constexpr (member_count<T>() == 1) {
        auto&& [p1] = t;
        if constexpr (Idx == 0) return &p1;
    } else if (member_count<T>() == 2) {
        auto&& [p1, p2] = t;
        if constexpr (Idx == 0) return &p1;
        if constexpr (Idx == 1) return &p2;
    }
    // ...
}
```

这招也是穷举法，重复写到一定数量，能够满足日常需求即可。只有等 C++26 Pack structure bindings 和 Pack indexing 真正可用后，才能从根本上解决该问题，到那时，以上代码便可以简化为：

```cpp
template<std::size_t Idx, typename T>
consteval auto get(T&& t)
{
    // pack structure bindings
    auto&& [...elems] = t;

    // pack indexing
    return &elems...[Idx];
}
```

这才是直击要害的解决方式，技巧到底不过是扬汤止沸之法。只是等到 C++26，反射大抵已入标准，这个场景下已没必要再自行实现获取成员名称了。

反射在本书的最后两章还会专门讲述，这里只是涉及其中的两个元函数而已，实现并不完善，先尝尝鲜罢。

6.9 Macros vs. Templates: A Generic Netlink Case Study

最后，再来看一个使用模板替代宏的实际场景。

这是一个关于 Generic Netlink（Linux 内核模块与用户空间应用之间结构化消息通信的一种方式）消息解析的例子，只展示用户空间解析内核传递消息的部分代码：

```
// Define policy for parsing attributes
static struct nla_policy demo_pol[GENLTEST_A_MAX + 1] = {
    [GENLDEMO_A_MSG]  = { .type = NLA_NUL_STRING },
    [GENLDEMO_A_LEN]  = { .type = NLA_U32 },
    [GENLDEMO_A_TYPE] = { .type = NLA_U8 }
};

// Handler for processing the reply from a netlink message
static int reply_handler(struct nl_msg *msg, void *arg) {
    nlattr* tb[GENLDEMO_A_MAX + 1];

    // Parse attributes from the netlink message
    int err = genlmsg_parse(nlmsg_hdr(msg),
        0, tb, GENLDEMO_A_MAX, demo_policy);
    if (error < 0) {
        std::cerr << "unable to parse message: "
            << strerror(-err);
        return NL_SKIP;
    }

    // Ensure the payload exists
    if (!tb[GENLDEMO_A_MAX]) {
        std::cerr << "msg attribute missing from message";
        return NL_SKIP;
    }

    char* msg;
    uint32_t len;
    uint8_t type;

    // Process attributes if present
    if (tb[GENLDEMO_A_MSG]) {
```

```
33          msg = nla_get_string(tb[GENLDEMO_A_MSG]);
34          return NL_SKIP;
35      }
36
37      if (tb[GENLDEMO_A_LEN]) {
38          len = nla_get_u32(tb[GENLDEMO_A_LEN]);
39          return NL_SKIP;
40      }
41
42      if (tb[GENLDEMO_A_TYPE]) {
43          len = nla_get_u8(tb[GENLDEMO_A_TYPE]);
44          return NL_SKIP;
45      }
46
47      return NL_OK;
48  }
```

reply_handler() 是一个回调函数，有消息到来时便会自动调用该函数处理。消息具有不同的类型，此处包含 GENLDEMO_A_MSG、GENLDEMO_A_LEN 和 GENLDEMO_A_TYPE 这三条内容，分别表示分别表示消息、消息长度和消息类型，类型分别为 NLA_NUL_STRING、NLA_U32 和 NLA_U8。

每种类型对应的解析函数并不相同，NLA_NUL_STRING 需要调用 nla_get_string() 解析，NLA_U32 需要调用 nla_get_u32() 解析，而 NLA_U8 则需要调用 nla_get_u8() 解析。C 中没有函数重载机制，因此看似相同的功能，却得借助不同名称的函数来处理。相似又有些许不同，便可以将变化分离出来，利用产生式元编程工具自动生成这些重复的代码。

若是纯 C 代码，宏便是唯一可用的产生式元编程工具，借其消除以上重复的方法如下：

```
1  #define NLA_GET_VALUE(attrs, attr, result, type) \
2      if (!attrs[attr]) { \
3          std::cerr << "No attr: " << #attr << "\n"; \
4          return NL_SKIP; \
5      } \
6      result = nla_get_##type(attrs[attr])
7
```

```
// Handler for processing the reply from a netlink message
static int reply_handler(struct nl_msg *msg, void *arg) {
    // ...

    char* msg;
    uint32_t len;
    uint8_t type;

    // Process attributes if present
    NLA_GET_VALUE(tb, GENLDEMO_A_MSG, msg, string);
    NLA_GET_VALUE(tb, GENLDEMO_A_LEN, len, u32);
    NLA_GET_VALUE(tb, GENLDEMO_A_TYPE, type, u8);

    // ...
}
```

通过定义一个 NLA_GET_VALUE 宏函数，抽离公共点，将每个变化点作为参数，利用宏的拼接和替换功能，自动生成了重复的代码。

这个需求只需用到宏的初级特性，没甚要点，不多絮烦。缺点很明显，一个“乱”字便可概括，也不存在类型安全，不知晓宏的定义和每种类型解析函数的名称，很难正确传递参数。

若是 C++ 代码，便可以利用模板作为产生式元编程工具来替代古老的宏，增强代码的简洁性和健壮性。实现变为：

```
using nla_u8     = uint8_t;
using nla_u16    = uint16_t;
using nla_u32    = uint32_t;
using nla_u64    = uint64_t;
using nla_string = char*;

class nla_parser {
public:
    explicit nla_parser(nlattr** tb) : tb_{tb} {}

    template<typename T>
    std::optional<T> get_value(unsigned int attr) const {
```

```
        if (!tb_[attr])
            return std::nullopt;

        if constexpr (std::is_same_v<T, nla_string>)
            return nla_get_string(tb_[attr]);
        else if constexpr (std::is_same_v<T, nla_u8>)
            return nla_get_u8(tb_[attr]);
        else if constexpr (std::is_same_v<T, nla_u16>)
            return nla_get_u16(tb_[attr]);
        else if constexpr (std::is_same_v<T, nla_u32>)
            return nla_get_u32(tb_[attr]);
        else if constexpr (std::is_same_v<T, nla_u64>)
            return nla_get_u64(tb_[attr]);
        else
            return std::nullopt;
    }

private:
    nlattr** tb_;
};

// Handler for processing the reply from a netlink message
static int reply_handler(struct nl_msg *msg, void *arg) {
    // ...

    // Process attributes if present
    nla_parser parser(tb);
    auto msg  = parser.get_value<nla_string>(GENLDEMO_A_MSG);
    auto len  = parser.get_value<nla_u32>(GENLDEMO_A_LEN);
    auto type = parser.get_value<nla_u8>(GENLDEMO_A_TYPE);
    if (!msg || !len || !type) {
        std::cerr << "Attributes missing from the message\n";
        return NL_SKIP;
    }

```

```
48      // ...
49  }
```

利用模板，转而定义一个 `nla_parser` 类，将每次重复检查的数据 `tb` 封于其中，便避免了宏中每次都要传递这个参数的必要。再在其中定义一个成员函数模板 `get_value()`，以模板参数抽离类型变化点，以函数参数抽离属性变化点，以返回值抽离结果变化点，所有变化俱已抽象。次再针对不同类型定义顾名思义的类型别名，利用 `if constexpr` 分发类型调用，返回更优的 `std::optional` 类型，一切便已成型。

这种方式远优于宏，一来变化点封装的逻辑更加清晰，不像宏只能统统作为参数，二来类型更加安全，未定义类型将返回 `std::nullopt`，三来调用流程更加自然，不必提前声明返回变量，四来错误处理可以自定义。

因此，宏和模板如何选择？规则其实相当简单，只有在迫不得已的情境下才使用宏，有选择的情况下，优先使用模板和其他更高级的产生式元编程工具。

6.10 C++26: Pack Indexing

当前，若要定义一个参数包变量，我们需要借助 `std::tuple`；若要索引式访问参数包元素，需要借助 `std::get` 和 `std::tuple_element`；若要解包，需要借助 `std::apply`。这些对于参数包的操作方式，只是当时情况下的权宜之计，并非最理想的解决方式。

因此，一直有许多关于模板参数包的更加直接的特性在提，比如 Pack declarations、Pack slicing 和 Pack literals，Pack Indexing 就是其中之一，它已经入了 C++26，所以本节讨论一下。

借助新的参数包特性，以后可以直接写出这样的代码：

```
1  template<typename... Ts>
2  class Tuple {
3  public:
4      constexpr Tuple(Ts&&... ts)
5          : elems(std::forward<Ts>(ts))...
6      { }
7
8      template<size_t I>
9      auto get() const& -> Ts...[I] const& {
10         return elems...[I]; // pack indexing
```

```
    }

private:
    Ts... elems; // variable packs
};

template<size_t I, typename... Ts>
struct std::tuple_element<I, Tuple<Ts...>> {
    using type = Ts...[I]; // pack indexing
};

int main() {
    Tuple<int, char> tup(1, 'c');
    return tup.get<0>();
}
```

这种实现 tuple 的方式借助了 Pack indexing 和 Variable packs(尚未入标准)，它比第九章和第十章将要介绍反射手法还要直接了当，是最简洁的实现方式。

归根到底，其他方式都没有釜底抽薪地解决参数包操作的根本问题，实现起来需要借助诸多技巧，非常麻烦。对于这些麻烦的方式，不应习以为常，也不应会点奇技淫巧就忽略了真正的问题。这是 C++ 历史的局限，最初就应该是这种直接了当的设计。

6.10.1 Syntax

深思熟虑过后，最终 Pack Indexing 的语法为：

```
name-of-a-pack ... [constant-expression]
```

这使得我们可以直接访问指定位置的参数包，例子：

```
template<typename... T>
constexpr auto first_plus_last(T... values) -> T...[0] {
    return T...[0](values...[0] +
        values...[sizeof...(values)-1]);
}

static_assert(first_plus_last(1, 2, 10) == 11);
```

`T...[N]` 针对的是 Types，而 `values...[N]` 针对的是 Values。参数包的首位元素和末位元素被相加起来，返回一个编译期常量值。

6.10.2 Limited Support

虽然 Pack Indexing 已入 C++26，但当前并未全部完善。

比如不支持 From-the-end-indexing，原本想用负数索引来表示从后向前访问，但可能会存在问题。看如下例子：

```
// Return the index of the first type convertible to Needle
// in Pack or -1 if Pack does not contain a suitable type.
template <typename Needle, typename... Pack>
auto find_convertible_in_pack;

// if find_convertible_in_pack<Foo, Types...> is -1,
// T will be the last type, erroneously.
using T = Types...[find_convertible_in_pack<Foo, Types...>];
```

`find_convertible_in_pack` 返回值若为 -1，则会导致语义错误。后面会解决这个问题，或是采用其他的语法形式，比如：

```
using Bar = T...[^1];    // C#. first from the end
using Bar = T...[$ - 1]; // Dlang. first from the end
```

支持容易，关键是要全面考虑潜在的问题。

还有下面这种简化语法尚不支持：

```
void g(auto&&);

template<typename...T>
void f(T&&... t) {
    // current proposal
    g(std::forward<T...[0]>(t...[0]));
    // not proposed nor implemented
    g(std::forward<T>(t)...[0]);
}
```

还有其他潜在冲突，在此不一一列举，待更加完善，再单独论述。

6.10.3 Future

Pack Indexing 只是向前走出了一小步，还有其他相关特性与其相辅相成，只有它们都进入标准，才能真正简化参数包的相关操作。

例如，Variable packs，允许直接定义一个参数包变量：

```
template<typename... Ts>
struct S {
    Ts... packs;
};
```

再如，Adding a layer of packness，允许将 tuple-like Types 转换为 Packs：

```
void f(std::tuple<int, char, double> t) {
    // equivalent to
    // g(std::get<0>(t), std::get<1>(t), std::get<2>(t))
    // or, possibly, std::apply(g, t)
    g(t.[:]...);

    // decltype(u) is the same as T - just a really
    // complex way to get there
    using T = decltype(t);
    std::tuple<T::[:]...> u = t;
}
```

`v.[I]` 和 `T::[I]` 分别是 `v.[:]...[I]` 和 `T::[:]...[I]` 的语法糖，省略索引，则相当于指定所有元素。

又如，Pack slicing，在以上特性的基础上，允许指定索引范围：

```
void h(std::tuple<int, char, double> t) {
    // a is a tuple<int, char, double>
    auto a = std::tuple(t.[:]...);

    // b is a tuple<char, double>
    auto b = std::tuple(t.[1:]...);

    // c is a tuple<int, char>
    auto c = std::tuple(t.[:-1]...);
```

```
10
11      // d is a tuple<char>
12      auto d = std::tuple(t.[1:2]...);
13  }
```

还有 Pack literals，允许直接创建一个参数包：

```
1  // b is a pack of ints
2  auto... b = { 1, 2, 3 };
```

可以用来为参数包设置默认参数：

```
1  template<typename... Ts = ...<int>>
2  void foo(Ts... ts = ...{0});
3
4  foo(); // calls foo<int>(0);
```

……

6.11 Summary

本章与第五章相辅相成，全面回顾了模板的理论和诸多妙技。篇幅有限，简单技术并未单独讨论，重心皆在高级技术上面。

妙艺有新有旧，实现却悉为新的手法，去除了过去的一些组件，转而利用标准已有组件，减少了代码重复。任何产生式技术，不论是宏还是模板，遍历都是核心技术，也是本章讲解的核心，递归和迭代，皆有优劣及使用场景。递归继承是模板代码生成的巅峰，是最复杂也是最精妙的技术，线性递归继承和散乱递归继承，一个实现逻辑，一个实现接口，一个是具体实现，一个是抽象表达，组合起来，能够实现复杂的代码注入功能。编译期获取结构体成员个数和名称是静态反射中的两个小小元函数，现有技术早能实现，只是仍旧复杂一些，代码重复难以尽除，这是底层特性的缺失，非应用层实现所能根除。编译期消息分发是字符串 NTTP 的一个巧绝应用，展示了一些关键技术，后续依旧有用。

总而言之，模板的顶端技术，本章几近一网全收，可反复阅读。

Chapter 7

Fold Expressions

模板是第一阶段元编程最核心的工具，中篇以两章五星难度的内容开头，深入纵览其核心技术与诸般妙诀。本章要讨论的元编程工具——Fold Expressions，依旧处于第一阶段，是 C++17 引入的一个跨越性产生式特性。

该特性从更高层面抽象了参数包拆解方式，消除了传统递归所带来的复杂性，是非常有用的编译期遍历方式。因此，这个特性是产生式元编程中的关键部分，这也是单独分配一章深度讨论的原因。

这个特性也并不完善，C++26/29 依旧会增加与其相关的特性，增加哪些？为什么加？痛点在哪儿？这些问题也将在本章给出答案。

7.1 Fold

Fold 这个概念来自函数式编程，本身指的是一类高阶函数，这些函数使用给定的组合操作来分析递归数据结构，并通过递归处理其组成部分的结果来重新组合，最终构建出一个返回值。简而言之，Fold 能够递归地遍历数据结构中的每个元素，并通过一个组合函数将这些元素的值合并为一个结果。这个组合函数通常定义了如何将两个值合并在一起，以及如何处理基础情况（如空数据结构）。

这种从递归数据结构中提取信息的数学概念称为 Catamorphism（源自古希腊语：κατά “向下” 和 μορφή “形式，形状”），表示从一个初始代数到其他代数的唯一同态映射。编程中，目标代数通常就是一个单一的值或结果。若从词的本意来理解，其实指的就是将一复杂数据结构向下拆解成简单数据结构的过程，这个过程递归地利用一个组合函数来完成。Catamorphism 通过抽象化递归数据结构的处理方式，提供了一种通用的模式来遍历和处理递归数据结构，使得代

码更具有可读性和可维护性，适用于各种场景，如求和、乘积、查找、过滤等。

Fold 只是 Catamorphism 的一个具体实现，主要是指对列表和序列的处理。例如，对列表 `[1, 2, 3, 4]` 求和，可以这样表示：

```
foldl (+) 0 [1, 2, 3, 4] = (((0 + 1) + 2) + 3) + 4 = 10
foldr (+) 0 [1, 2, 3, 4] = 1 + (2 + (3 + (4 + 0))) = 10
```

想必大家也不陌生，`foldl` 是从左向右应用函数的方式，称为左折叠，`foldr` 与之相反，称为右折叠。若是操作类型满足交换律，不论选择哪种方式，结果都一样。由此结构，也可以看到 Fold 通常包含的三个参数：

- 二元函数：`+`，定义了如何将列表的两个元素合并成一个值；
- 初始值：`0`，折叠操作的开始值，它与列表的第一个元素一起传递给二元函数；
- 列表：`[1, 2, 3, 4]`，折叠操作的元素集合。

当然，倘是一元函数，则不需要初始值，例如，拼接字符串：

```
foldl (++) ["a", "b", "c", "d"] = ((("a" ++ "b") ++ "c") ++
↪  "d")) = "abcd"
foldr (++) ["a", "b", "c", "d"] = "a" ++ ("b" ++ ("c" ++
↪  "d")) = "abcd"
```

正是这种更高一级的抽象方式，Fold 这个概念才能够简化常规的递归方式，以一种更加易于人类理解的方式表达拆解逻辑，增加代码的可读性的同时，也简化了编码效率。

7.2 C++ Fold

`std::accumulate` 就是 C++ 提供的一个 Fold 函数，包含前面所说的三个参数。基本形式如下：

```
// left fold
std::accumulate(begin, end, initval, func)
// right fold
std::accumulate(rbegin, rend, initval, func)
```

下面是一个例子：

```cpp
// fr. https://en.cppreference.com/w/cpp/algorithm/accumulate
#include <functional>
#include <iostream>
#include <numeric>
#include <string>
#include <vector>

int main() {
    std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int sum = std::accumulate(v.begin(), v.end(), 0);
    int product = std::accumulate(v.begin(), v.end(), 1,
    ↪ std::multiplies<int>());

    auto dash_fold = [](std::string a, int b) {
        return std::move(a) + '-' + std::to_string(b);
    };

    std::string s = std::accumulate(
        std::next(v.begin()), v.end(),
        std::to_string(v[0]), // start with first element
        dash_fold);

    // Right fold using reverse iterators
    std::string rs = std::accumulate(
        std::next(v.rbegin()), v.rend(),
        std::to_string(v.back()), // start with last element
        dash_fold);

    std::cout << "sum: " << sum << '\n'
        << "product: " << product << '\n'
        << "dash-separated string: " << s << '\n'
        << "dash-separated string (right-folded): " << rs;
}
```

输出为：

```
sum: 55
product: 3628800
dash-separated string: 1-2-3-4-5-6-7-8-9-10
dash-separated string (right-folded): 10-9-8-7-6-5-4-3-2-1
```

函数刚好接受一个列表、一个初始值、一个二元定制函数作为参数，所以类似的需求皆可摆脱传统的遍历方式，表达起来更加简单。但是 `std::accumulate` 只支持二元函数，且表意不够广泛，因而 C++23 又增加了 Ranges fold 系列算法。基本形式如下：

```
std::ranges::fold_left(range, initval, func)
std::ranges::fold_right(range, initval, func)
std::ranges::fold_left_first(range, func)
std::ranges::fold_right_last(range, func)
```

新的系列算法更加顾名思义，同时支持一元函数和二元函数，进一步简化了折叠方式。

同样提供一个例子：

```
// fold algorithms
int xs[] = { 1, 2, 3, 4, 5 };
auto concatl = [](std::string s, int i) {
    return s + std::to_string(i);
};
auto concatr = [](int i, std::string s) {
    return s + std::to_string(i);
};

auto fold_left  = ranges::fold_left(xs, std::string(),
↪  concatl);
fmt::print("fold left: {}\n", fold_left);

auto fold_right = ranges::fold_right(xs, std::string(),
↪  concatr);
fmt::print("fold right: {}\n", fold_right);

```

```
// Output:
// fold left: 12345
// fold right: 54321
```

7.3 C++ Fold Expressions

Fold 函数主要适用于列表或其他线性数据结构，而 C++17 Fold Expressions 则适用于可变模板参数包。不同的是，前者是 Library 级别的特性，而后者却是 Language 级别的特性，可用性更强。

Fold Expressions 同样支持一元及二元的左折叠和右折叠，形式如下：

```
( pack op ... )          // Unary right fold
( ... op pack )          // Unary left fold
( pack op ... op init )  // Binary right fold
( init op ... op pack )  // Binary left fold
```

逻辑其实都一样，只是语法形式稍异而已。... 在参数包的左边，就属于左折叠，在右边，就属于右折叠。但是，在二元折叠中，不能同时包含参数包，例如：

```
// fr. ISO/IEC 14882:2020 §7.5.6
template<typename ...Args>
bool f(Args ...args) {
    // OK
    return (true && ... && args);
}

template<typename ...Args>
bool f(Args ...args) {
    // error: both operands contain unexpanded packs
    return (args + ... + args);
}
```

而在一元折叠中，参数包若为空，只有表 7.1 中的三个操作符具有合法的默认值，这三个操作符也恰恰是各种高级技巧的基石。

Table 7.1: Value of folding empty sequences

Operator	Value when parameter pack is empty
&&	true
||	false
,	void()

7.4 Smart Tricks with Fold Expressions

Fold 是更加抽象化的遍历方式，Fold Expressions 的核心作用就是替代传统的递归遍历方式。但是，要精细化控制这种遍历方式的各种细节，例如条件、中断、中间值、下标等，便需要各种高级技巧了。

本节便分别展示各种小技巧，将它们分布在各个算法当中。

7.4.1 Conditions and Counting

根据一个 Predicate 函数，计算符合条件的元素个数。

`all_of` 计算是否所有元素都满足条件，`any_of` 计算是否任一元素满足条件，`count_of` 计算满足条件的个数。实现如下：

```
// Check whether all elements matches a predicate.
auto all_of(auto F, auto... args) -> bool {
    return (F(args) && ...);
}

// Check whether any elements matches a predicate.
auto any_of(auto F, auto... args) -> bool {
    return (F(args) || ...);
}

// Count the elements matches a predicate.
auto count_of(auto Pred, auto... args) -> int {
    return (Pred(args) + ...);
}
```

标准中存在类似的算法给容器使用，实现都采用 `std::find_if` 之类的算法，查找算法内部又都涉及传统的循环遍历。对于参数包，若是用传统的递归来完成此类操作，相较也会麻烦，而 Fold Expressions 这种更高一级的遍历方式则显得灵活而简洁。

本技巧主要利用了两个特性，一个是 `&&` 和 `||` 所具有的 *short-circuit* 评估能力，可以用来实现条件和中断，另一个是 `bool` 到 `int` 之间所存在的隐式转换，可以用来计数。

7.4.2 Random Access

任意访问参数包某个索引指向的元素。

首先，若是参数包的元素属于同构类型，可以通过以下方式访问其首尾元素。

```
// Find the first element.
auto first_of(auto... args)
    -> std::common_type_t<decltype(args)...> {
    std::common_type_t<decltype(args)...> result;
    ((result = args, true) || ...);
    return result;
}

// Find the last element.
auto last_of(auto... args)
    -> std::common_type_t<decltype(args)...> {
    std::common_type_t<decltype(args)...> result;
    (result = (args, ...));
    return result;
}
```

本处技巧主要是利用 `||` 和 `,` 的特性，保存中间值的过程中，决定是否继续往下走。同时，还用到 `=` 让右边的表达式先计算，从而保存最终结果。

其次，若是参数包是异构类型，可以借助 `std::tuple` 的索引式访问算法来实现，返回值只能依靠自动推导。

```
auto generic_first_of(auto... args) {
    auto values = std::forward_as_tuple(args...);
    return std::get<1>(values);
}

auto generic_last_of(auto... args) {
```

```
    auto values = std::forward_as_tuple(args...);
    return std::get<sizeof...(args)-1>(values);
}
```

这种方式本质就是利用已有的 `std::tuple` 算法来达到目的，虽说复杂度降低，但却要构造一个额外的对象。

最后，若是参数包的元素属于同构类型，不借助 `std::tuple`，可以通过以下方式实现索引式访问。

```
// Find the nth element.
template <std::size_t I>
auto nth_of(auto... args)
    -> std::common_type_t<decltype(args)...> {
    std::common_type_t<decltype(args)...> result;
    std::size_t n{};
    ((n++ == I ? (result = args, true) : false) || ...);
    return result;
}
```

手法结合了前面几个技巧，再以三目运算符作为条件，分发逻辑，`false` 时继续往下遍历，`true` 时结束遍历。

7.4.3 Maximum and Minimum

取同构元素列表的最大最小值。

同样需要保存中间结果，但不需要中断遍历流程，实现如下：

```
// Find the minimum element.
auto min_of(auto... args)
    -> std::common_type_t<decltype(args)...> {
    auto min = (args, ...);
    ((min > args ? min = args : 0), ...);
    return min;
}

// Find the maximum element.
auto max_of(auto... args)
```

```
    -> std::common_type_t<decltype(args)...> {
    auto max = (args, ...);
    ((max < args ? max = args : 0), ...);
    return max;
}
```

不中断的情况下，使用 , 展开更加便捷。

7.4.4 Reverse Packs

逆转列表元素位置，返回转换后的列表。

因为需要返回一个列表，所以只得借助 `std::tuple`，再反转 `std::tuple`。实现如下：

```
auto reverse_of(auto... args) {
    auto tuple = std::make_tuple(args...);
    return [&tuple]<auto... I>(std::index_sequence<I...>) {
        return std::make_tuple(std::get<sizeof...(args)-1-I>(
            std::forward<decltype(tuple)>(tuple))...);
    }(std::index_sequence_for<decltype(args)...>{});
}
```

此处便出现了第六章介绍的 Compile-Time for，这种技巧可以在编译期遍历并操作 `std::tuple`，本质就是创造一个索引参数包，再以 Fold Expressions 遍历。

7.5 Overload Pattern

与 Fold Expressions 相关联的另一个技术称为 Overload Pattern，这项技术通过展开参数包实现多继承，以在视觉层面模拟 Lambda 重载。代码只有两行：

```
template<class... Ts>
struct overloaded : Ts... { using Ts::operator()...; };
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;
```

短短两行代码中，除了使用参数包展开实现多继承，还借助 Using-declration 来绕开重载合并规则，避免重载歧义。这些特性和 Fold Expressions 类似，都能展开参数包，生成重复代码。

Lambda 重载是一种灵活的定制方式，以前展示过对象工厂和抽象工厂的应用，下面看一个与 Fold Expressions 一起使用的新例子。

7.6 Implement any_visit with Fold Expressions

`std::any` 采用类型擦除技术，实现了异构类型表示，可以和容器类型结合起来构成异构容器。`std::variant` 也能够起到类似的作用，标准提供 `std::visit` 来访问元素，因此可以直接使用已有算法来遍历：

```
using value_type = std::variant<int, double, std::string>;
std::vector<value_type> container;
container.push_back(5);
container.push_back(0.42);
container.push_back("hello");

// Iterate the heterogeneous container
std::ranges::for_each(container, [](const value_type& value)
↪ {
    std::visit([](const auto& x){ std::print("{} ", x); },
↪ value);
});
```

而 `std::any` 没有对应的 `visit` 访问函数，只能通过下面这种 type-switch 方式访问：

```
for (const auto& a : container) {
    if (a.type() == typeid(int)) {
        const auto& value = std::any_cast<int>(a);
    } else if (a.type() == typeid(const char*)) {
        const auto& value = std::any_cast<const char*>(a);
    } else if (a.type() == typeid(bool)) {
        const auto& value = std::any_cast<bool>(a);
    }
}
```

重复、变化、繁琐……于是实现一个 `any_visit` 的想法顿时出现，而这个遍历就可以用 Fold Expressions 来抽象得更高一级。实现为：

```
template <class... Ts>
void any_visit(auto f, const std::any& a) {
    ((std::type_index(a.type()) ==
    ↪ std::type_index(typeid(Ts))
        && (f(std::any_cast<Ts>(a)), true)) || ...);
}
```

可以细品一下这个实现是如何借助 `&&`、`||` 和 `,` 消除 `for` 和 `if` 的，技巧都是前面讲过的内容。有了这个工具，便可以像 `std::visit` 那样，借助算法来迭代 `std::any` 构成的异构容器。例子：

```
std::vector<std::any> container { 5, 0.42, "hello", false };
// Output: 5 0.42 hello boolean: false
std::ranges::for_each(container, [](const auto& a) {
    any_visit<int, double, const char*>(
        [](const auto& x) { std::print("{} ", x); }, a);
    any_visit<bool>(
        [](const auto& x) { std::print("boolean: {} ", x); },
    ↪ a);
});
```

简单是简单，但由于 Fold Expressions 要借助参数包展开，模板参数的变化性依旧没有消除，而这些信息其实可以表现到 Lambda 参数之中，以 Overload pattern 封装这部分变化。即目标用法变成这样：

```
std::ranges::for_each(container, [](const auto& a) {
    any_visit(overloaded {
            [](int x) { std::print("int: {} ", x); },
            [](double x) { std::print("double: {} ", x); },
            [](std::string_view x) { std::print("string: {}
    ↪ ", x); },
            [](bool x) { std::print("bool: {} ", x); }
        }, a);
    });
```

如此一来，不仅可以精确处理每一个异构类型，还不用重复调用 `any_visit`，同时也消除了显式模板参数。`overloaded` 在上一节已然介绍，那么现在只剩下

一个关键问题——如何获取 Lambda 的参数类型？解决了这个问题，实现便也水到渠成了。

Lambda 是一个可以携带状态的函数，其实现是一个含有 `operator()` 重载的匿名类，捕获的参数作为匿名类的数据成员直接初始化。Lambda 使用时调用的便是这个重载的 `operator()`，返回的类型就是匿名类的类型，称为 closure type。因此，问题进一步转化为如何获取成员函数 `operator()` 的参数类型，通过第五、第六章的高级模板内容，获取起来犹如探囊取物。

方法就是把想要的类型，通过模板参数，显式写出来：

```
// For a function pointer
template<typename R, typename Arg, typename... Rest>
Arg extract_first_arg(R(*) (Arg, Rest...));

// For a member function pointer without a qualifier
template<typename R, typename F, typename Arg, typename...
↪  Rest>
Arg extract_first_arg(R(F::*) (Arg, Rest...));

// For a const-qualified member function pointer
template<typename R, typename F, typename Arg, typename...
↪  Rest>
Arg extract_first_arg(R(F::*) (Arg, Rest...) const);

// ...
```

这里只写了支持函数指针和带基本修饰的 Lambda 函数，更多修饰可以接着往下写。这些函数都属于稻草人函数，不实际使用，只提取类型，故无需实现。接着，通过 `decltype()` 将函数模板的返回类型提取出来：

```
template<typename L>
using lambda_arg_t =
↪  decltype(extract_first_arg(&L::operator()));
```

至此，最复杂的问题便解决了。

最后，利用 Overload pattern 和 Fold expressions 访问 `std::any` 构成的异构容器。代码为：

```
template<class... Ts>
struct overloaded : Ts... { using Ts::operator()...; };
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;

template<typename... Lam>
void any_visit(const overloaded<Lam...>& f, auto&& any) {
    ((std::type_index(any.type()) ==
    ↪ std::type_index(typeid(lambda_arg_t<Lam>))
        && (f(std::any_cast<lambda_arg_t<Lam>>(any)), true))
    ↪ || ...);
}
```

寥寥数行代码，便解决了一个相对复杂的问题，这就是 Fold expressions 的威力。

7.7 Related Features and Discussions

Fold expressions 是专门针对参数包的 Fold 特性，好用是好用，但是前提得先有参数包，否则也是巧妇难为无米之炊。

比如我们要循环输出 5 次 hello fold expressions，那前提是先得有五个模板参数，在哪儿凭空产生固定个数的模板参数呢？这其实就是第六章所介绍的 Compile-time for 技术，采用该技术，可以这样实现：

```
[]<auto... Is>(std::index_sequence<Is...>) {
    ((std::println("hello fold expressions"), Is), ...);
}(std::make_index_sequence<5>{});
```

这能达到预期效果，但显得复杂，更本质的问题在于 C++ 缺少直接创建参数包的能力，`std::index_sequence` 也只是扬汤止沸的产物，没有解决根本问题。而 Circle 便支持直接创建参数包，于是可以简洁地完成以下功能：

```
struct obj_t {
  int x;
  double y;
  std::string s;
};

```

```
int main() {
  auto f = [](const char* name, auto... i) {
    std::cout<< name<< ":\n";
    std::cout<< "  "<< i<< "\n" ...;
  };

  // Just expand a pack into an argument list.
  f("integers", int...(5) ...);

  obj_t obj { 100, 3.14, "A string" };
  f("object", obj...);
}

// Compiler Explorer: https://godbolt.org/z/4K61vM9hf
```

`int...(5) ...` 可以直接创建一个 `int` 型参数包并展开，`obj...` 可以直接将结构体的成员转换为参数包。有了这种创建参数包的能力，不但能够简化代码，而且可以直接折叠结构体，极大增强操纵模板参数的能力。

不过，随着 C++26 Pack structure bindings 和 Pack indexing 的加入，在一定程度上能够改善这部分问题。那时 Compile-time for 便无需借助 Lambda 充当辅助函数了，可以直接这样写：

```
auto [...Is] = std::make_index_sequence<5>{};
((std::println("hello fold expressions"), Is), ...);
```

这才是最直接的方式，Fold expressions 如今主要就受限于参数包的特性不足。

7.8 Ternary Right Fold Expression

Fold expressions 在遍历时，还缺少一种处理错误的方式，看如下例子：

```
template<std::size_t... Is>
auto test_impl(std::size_t j, std::index_sequence<Is...>) {
    return ((j == Is ? (std::println("found"), true) : 0) ||
↪   ...);
}
```

```
5
6  template<std::size_t N>
7  auto test(std::size_t j) {
8      return test_impl(j, std::make_index_sequence<N>{});
9  }
10
11 int main() {
12     test<5>(5);
13 }
```

当遍历查找失败时，Fold expressions 无法处理错误。而 Ternary Right Fold Expression 就是解决这个问题的，代码变成：

```
1  template<std::size_t... Is>
2  auto test_impl(std::size_t j, std::index_sequence<Is...>) {
3      return ((j == Is ? std::println("found")
4          : ... : throw std::range_error("Out of range"));
5  }
```

可以看到，这个特性可以消除 || 和 ,，进一步简化代码，并且在查找失败时，可以处理错误。

总结一下，它的语法格式是这样的：

```
( C ? E : ... : D )
```

展开变为：

```
( C(1) ? E(1) : ( ... ( C(N-1) ? E(N-1) : ( C(N) ? E(N) : D )
↪  ) ) )
```

这个特性可能会入 C++26。

7.9 Best Practice: Convert Run-Time Values to Compile-Time Constants

本节源于 “水滴会飞” 的一个提问，需求略显特别，却是一个非常不错的需要使用 Fold Expressions 的产生式元编程例子。

7.9.1 Original Problem

优化后的原始问题，代码如下：

```
struct base {
    virtual void foo() = 0;
};

template<int I, int J, int K, int L>
struct derived : public base {
    void foo() override final {
        std::cout << "derived<" << I << ',' << J << ','
            << K << ',' << L << ">::foo()\n";
    }
};

template<int I, int J, int K, int L>
base* create_instance_impl() {
    return new derived<I, J, K, L>();
}

base* create_instance(int i, int j, int k, int l) {
    // Error, i, j, k and l are not constant expressions.
    return create_instance_impl<i,j,k,l>();
}

int main() {
    create_instance(0,0,0,0)->foo();
    create_instance(0,1,2,3)->foo();
}
```

简而言之，某些原因下，可能是老项目，也可能是外部依赖，类的原始设计需要编译期的值作为模板参数，而如今这些输入却只能在运行期获得（从文件读、从网络接收……），不能更改原有类的情况下，只有将这些运行期的值转换成编译期值，才能正常调用。

运行期的值如何能转换成编译期？

它们一个发生于运行期，一个发生于编译期，似乎没有办法。但是，本质

上还是一种 Value-to-Type 的问题，在实现 Factory 模式时，便系统探索过这个问题的解决方案，核心思路就是映射，针对每个值，一一提供 Value-to-Type 的对应关系。

这个问题还隐藏着一个问题，数字之间其实暗含着一层组合关系，假设 $i/j/k/l$ 的最大取值分别为 $M_i/M_j/M_k/M_l$，则共有 $Mi \times Mj \times Mk \times Ml$ 种组合方式。

倘若按照普通方式映射，实现如下：

```
base* create_instance(int i, int j, int k, int l) {
    if (i == 0 && j == 0 && k == 0 && l == 0)
        return create_instance_impl<0, 0, 0, 0>();
    else if (i == 0 && j == 0 && k == 0 && l == 1)
        return create_instance_impl<0, 0, 0, 1>();
    else if (i == 0 && j == 0 && k == 0 && l == 2)
        return create_instance_impl<0, 0, 0, 2>();
    // Add more cases for other (i, j, k, l) combinations as
    ↪ needed
}
```

可见，由于组合数量太多，根本不可能写尽映射。因此，问题的一个关键便在于减少映射数量。

7.9.2 Dimensionality Reduction: Manual Approach to the Solution

数学中，乘法是一种能够为稀少信息升维的工具，而加法则是一种降维的工具。若是能够将 $Mi \times Mj \times Mk \times Ml$ 变成 $Mi + Mj + Mk + Ml$，势必会极大地减少映射数量。

怎么将乘法转变为减法呢？

关键在于把立体结构转换成线性结构，当前 $i/j/k/l$ 是一种交织在一起的高维结构，我们需要分散它们，将它们从高维降至一维。

这种降维思路下的映射方式，代码如下：

```
template<int I, int J, int K>
base* create_instance_of_k(int l) {
    if (l == 0)
        return create_instance_impl<I, J, K, 0>();
    else if (l == 1)
```

```
6           return create_instance_impl<I, J, K, 1>();
7       else if (l == 2)
8           return create_instance_impl<I, J, K, 2>();
9       else if (l == 3)
10          return create_instance_impl<I, J, K, 3>();
11      else
12          return nullptr;
13  }
14
15  template<int I, int J>
16  base* create_instance_of_j(int k, int l) {
17      if (k == 0)
18          return create_instance_of_k<I, J, 0>(l);
19      else if (k == 1)
20          return create_instance_of_k<I, J, 1>(l);
21      else if (k == 2)
22          return create_instance_of_k<I, J, 2>(l);
23      else
24          return nullptr;
25  }
26
27  template<int I>
28  base* create_instance_of_i(int j, int k, int l) {
29      if (j == 0)
30          return create_instance_of_j<I, 0>(k, l);
31      else if (j == 1)
32          return create_instance_of_j<I, 1>(k, l);
33      else if (j == 2)
34          return create_instance_of_j<I, 2>(k, l);
35      else
36          return nullptr;
37  }
38
39  base* create_instance(int i, int j, int k, int l) {
40      if (i == 0)
```

```
41          return create_instance_of_i<0>(j, k, l);
42      else if (i == 1)
43          return create_instance_of_i<1>(j, k, l);
44      else if (i == 2)
45          return create_instance_of_i<2>(j, k, l);
46      else
47          return nullptr;
48  }
```

此处，$M_i/M_j/M_k/M_l$ 分别取 $2/2/2/3$，所以降维前共有 $3\times3\times3\times4=108$ 种映射关系，而降维后仅有 $3+3+3+4=13$ 种映射关系。

通过降维法，成功将一个无法穷尽的映射关系减少到能够穷尽的数量，从而解决了本问题。

但需要分清，降维法只是降低了需要手动编写的映射数量，并非是改变组合的数量。Value-to-Type 必须一一对应，实际上生成的模板特化依旧有 108 种。

因此，问题规模从一开始就需要定好上界，否则模板爆炸式生成，编译时间也会爆炸式增长。

7.9.3 Dimensionality Reduction: Automated Approach to the Solution

手动版增加映射关系需要写很多代码，可以利用本章和前面几章介绍的各种技巧实现自动生成映射关系。

核心实现如下：

```
1   /**
2   * Converts runtime values to compile-time constants and
    ↪  invokes the callback function `call`.
3   *
4   * Template Parameters:
5   * - ReturnType: The return type of the callback function
    ↪  `call`.
6   * - T: The type of the runtime variables to be converted into
    ↪  compile-time constants.
7   * - Is...: A pack of compile-time constants derived from the
    ↪  runtime values.
```

```
8  * - F: The type of the callback function that will be invoked
   ↪  with the compile-time values.
9  * - Args...: The types of the remaining runtime variables to
   ↪  be converted into compile-time constants.
10 * - FirstLimit: The maximum value for the first runtime
   ↪  variable.
11 * - Limits...: The maximum values for the remaining runtime
   ↪  variables.
12 *
13 * Function Parameters:
14 * - call: A callable object (e.g., lambda or function) that
   ↪  will be invoked with the compile-time constants.
15 * - first: The first runtime variable, which will be
   ↪  converted to a compile-time constant.
16 * - args...: The remaining runtime variables, each of which
   ↪  will also be converted into compile-time constants.
17 *
18 * Constraints:
19 * - The number of runtime variables (Args...) must equal the
   ↪  number of their respective maximum values (Limits...).
20 */
21 template<class ReturnType, class T, T... Is, class F,
22     class... Args, size_t FirstLimit, size_t... Limits>
23     requires (sizeof...(Args) == sizeof...(Limits))
24 ReturnType to_compile_time_values_impl(F&& call,
25     std::index_sequence<FirstLimit, Limits...>,
26     T first, Args&&... args) {
27     if constexpr (sizeof...(args) == 0) {
28         return [first, call = std::forward<F>(call)]
29             <auto... I>(std::index_sequence<I...>) {
30             if constexpr (std::is_same_v<ReturnType, void>) {
31                 ((first == I ?
32                     (call(std::index_sequence<Is..., I>{}),
   ↪  true)
33                     : false) || ...);
```

```
34              } else {
35                  ReturnType result{};
36                  ((first == I ?
37                      (result = call(std::index_sequence<Is...,
    ↪  I>{}), true)
38                      : false) || ...);
39                  return result;
40              }
41          }(std::make_index_sequence<FirstLimit>{});
42      } else {
43              if constexpr (std::is_same_v<ReturnType, void>) {
44                  [first, call = std::forward<F>(call),
45                  ...args = std::forward<Args>(args)]
46                  <auto... I>(std::index_sequence<I...>) {
47                      ((first == I ?
    ↪  (to_compile_time_values_impl<ReturnType, T, Is...,
    ↪  I>(std::move(call),std::index_sequence<Limits...>{},
    ↪  args...), true)
48                      : false) || ...);
49                  }(std::make_index_sequence<FirstLimit>{});
50              } else {
51                  return [first, call = std::move(call),
52                      ...args = std::forward<Args>(args)]
53                      <auto... I>(std::index_sequence<I...>) {
54                      ReturnType result{};
55                      ((first == I ? (result =
    ↪  to_compile_time_values_impl<ReturnType, T, Is...,
    ↪  I>(std::mov(call), std::index_sequence<Limits...>{},
    ↪  args...), true)
56                      : false) || ...);
57                      return result;
58                  }(std::make_index_sequence<FirstLimit>{});
59              }
60      }
61  }
```

除了满足需求，其中还处理了返回值问题，`call` 是一个泛型函数或泛型 Lambda，在真正调用之前，无法获取其返回类型，必须得通过模板显式传递该返回类型，而 `void` 无法声明结果变量，因此分了情况处理。抛开这些为了通用性而做的特殊处理，核心逻辑其实也就不到 20 行。

为了使用方便，外部接口也分了多种情况，实现如下：

```
/**
* Limits... represents the maximum values for the corresponding runtime variables.
* If these values are too large, the number of combinations will increase significantly,
* potentially causing a template instantiation explosion and greatly extending compilation time.
*/
template<class ReturnType,
    size_t... Limits, class F, class... Ints>
requires(sizeof...(Limits) <= sizeof...(Ints))
ReturnType to_compile_time_values(F&& call, Ints&&... ints) {
    if constexpr (sizeof...(Limits) == sizeof...(Ints)) {
        return to_compile_time_values_impl<ReturnType>(
            std::forward<F>(call),
            std::index_sequence<Limits...>{},
            std::forward<Ints>(ints)...);
    } else if constexpr (sizeof...(Limits) == 1) {
        constexpr std::size_t V = std::get<0>(std::tuple(Limits...));
        return to_compile_time_values_impl<ReturnType>(
            std::forward<F>(call),
            make_repeat_index_sequence<sizeof...(Ints), V>{},
            std::forward<Ints>(ints)...);
    } else {
        return to_compile_time_values_impl<ReturnType>(
            std::forward<F>(call),
            make_repeat_index_sequence<sizeof...(Ints), 5>{},
            std::forward<Ints>(ints)...);
    }
```

```
}
```

`Limits...` 表示每个变量的最大值，若是每个变量的最大值不同，则应该分别指定；若是每个变量的最大值相同，则只需指定一个；若是不指定，默认最大值皆为 5。

如前文所言，倘若 `Limits...` 的数量很多或是很大，映射的组合形式也会很大，从而导致模板爆炸，极大地增加编译时间。这里所说的“极大”并非是危言耸听，以四位数字为例，假设每位数字皆取 10，则 $10 \times 10 \times 10 \times 10$ 共有 10,000 种组合结果，编译器需要特化生成如此多的类，时间消耗，可想而知。

此外，`make_repeat_index_sequence` 是为了生成指定个数的相同索引，实现为：

```
template<size_t N, size_t V, size_t... Vs>
struct make_repeat_index_sequence_impl
    : make_repeat_index_sequence_impl<N-1, V, V, Vs...>
{};

template<size_t V, size_t... Vs>
struct make_repeat_index_sequence_impl<1, V, Vs...>
    : std::type_identity<std::index_sequence<Vs..., V>>
{};

template<size_t N, size_t V>
using make_repeat_index_sequence =
↪   make_repeat_index_sequence_impl<N, V>::type;
```

简简单单的辅助类，不需赘述。

7.9.4 Example: Generalized Automated Solution

该通用实现有一定的灵活性，可以直接将运行期值转换为编译期。

先来看一个简单的例子：

```
int i = 1;
int j = 2;
int k = 3;

// Output: 1 2 3
```

```
to_compile_time_values<void>(
    []<auto... Is>(std::index_sequence<Is...>) {
    ((std::cout << Is << " "), ...);
}, i, j, k);
```

对于原问题，借助该实现解决为：

```
base* create_instance(int i, int j, int k, int l) {
    return to_compile_time_values<base*>(
        []<auto... Is>(std::index_sequence<Is...>) {
        return create_instance_impl<Is...>();
    }, i, j, k, l);
}

int main() {
    // Output: derived<0,0,0,0>::foo()
    create_instance(0,0,0,0)->foo();

    // Output: derived<0,1,2,3>::foo()
    create_instance(0,1,2,3)->foo();
}
```

若是想效率更高一点，最好指定每个运行期值的最大范围，这样使用：

```
/**
* Creates an instance of `base*` by converting the runtime
↪  values i, j, k, l
* into compile-time constants.
*
* The maximum allowed values for the parameters are:
*  - i: 4
*  - j: 3
*  - k: 3
*  - l: 4
*
* These limits correspond to the template arguments of
↪  `to_compile_time_values<base*, 4, 3, 3, 4>`.
```

```
* Exceeding these values may result in undefined behavior or
↪ compilation failure.
*/
base* create_instance(int i, int j, int k, int l) {
    return to_compile_time_values<base*, 4, 3, 3, 4>(
        []<auto... Is>(std::index_sequence<Is...>) {
        return create_instance_impl<Is...>();
    }, i, j, k, l);
}
```

那么若是 $i/j/k/l$ 的值超过 $3/2/2/3$（下标从 0 开始，故减 1），将没有任何返回值：

```
int main() {
    // result is nullptr
    base* result = create_instance(0,1,2,4);
    if (result) {
        result->foo();
    }
}
```

如果 $i/j/k/l$ 的最大值都是 4，那么无需全部写出，只需写一个便可，如：

```
base* create_instance(int i, int j, int k, int l) {
    return to_compile_time_values<base*, 4>(
        []<auto... Is>(std::index_sequence<Is...>) {
        return create_instance_impl<Is...>();
    }, i, j, k, l);
}

int main() {
    // OK
    create_instance(0,0,0,0)->foo();
    create_instance(0,1,2,3)->foo();
}
```

这些灵活性，就是代码量增加的原因所在，再小的功能，只要考虑通用性，复杂性就会显著提升。

7.9.5 Summary

本节算是一个最佳实践，使用了诸多深入层次的产生式元编程技术，这些技术都在本章和前面几章中介绍过，因此这里不曾细致讲解实现细节。

涉及到的技巧很多，该实践甚至是一个模板爆炸的绝佳例子，组合数量过大，编译器都会直接编译超时。

同时，这里还介绍了一个利用乘法和加法进行升维降维的思路，也正是这个思路给了实现以理论支撑。

完整实现和示例：`https://godbolt.org/z/47E6fbqGr`。

7.10 Summary

本章更为全面而深入地介绍了 Fold Expressions，这是元编程的编译期遍历方式，能够消除传统递归和循环，简化代码，减少重复。

Fold 这个概念从函数式编程而来，是一种抽象等级更高的遍历函数，因此不同于传统面向对象和面向过程范式中的递归和循环遍历，表达起来更加自然。示例代码依旧非常丰富，展示了各种高级技巧，这些技巧用于提供条件、中断、中间值等能力。

Chapter 8

Constant Expressions

常量表达式能够在编译期产生值，当然也可以作为一种产生式元编程工具，为了区分其他工具，本书将此种方式称为常量表达式元编程。

常量表达式也是 C++ 元编程第二阶段发展所产生的核心工具，用来简化部分模板元编程的同时，也为反射元编程奠定了基础。因此，熟练掌握此种元编程方式，是迈入高级产生式元编程的必要条件，本章就来全面而深入地对此展开介绍。

闲言休叙，让我们直入主题吧！

8.1 A First Look at Constant Expressions

常量表达式的概念在 C++98 便已存在，只是较为模糊，仅限于简单的字面值常量（如 `1`, `2.5`, `true` 等）。一言以蔽之，常量表达式就是在编译期即可确定值的表达式。

在没有引入 `constexpr` 关键字之前，字面值、枚举值、`const` 变量、`sizeof` 表达式……也都属于常量表达式。一个小例子：

```
enum { index = 2 };

template<size_t> struct S {
    static const int value = 2;
};

void f() {
```

```
8       const int size = 5;
9       // ok, size is a constant expression
10      std::array<int, size> arr1;
11
12      // ok, 5 is a constant expression
13      const int arr2[5] = { 0, 1, 2, 3, 4 };
14
15      // ok, index is a constant expression
16      int arr3[index];
17
18      // ok, sizeof expression is a constant expression
19      S<sizeof(char)> s1;
20
21      int n = 4;
22      static int m = 3;
23      // error, n and m are not constant expressions
24      S<n> s2;
25      S<m> s3;
26
27      // ok
28      static_assert(S<2>::value == index);
29      // error, arr2 is not a constant expression
30      static_assert(arr2[1] == sizeof(char));
31  }
```

C++11 之后，每个 C++ 版本都在不断增强常量表达式的能力，`constexpr` 能够更为准确地声明一个能够用于编译期求值的变量和函数。不仅是单一的字面值，如今的常量表达式甚至能够编写复杂的逻辑，而这些编译期的代码和运行期的代码差异不大，于是可以用来替代模板元编程，实现一些逻辑清晰的编译期代码。

8.2 Templates vs. Constant Expressions

模板和常量表达式都是 C++ 提供的元编程机制，使用它们编写的代码都能够在编译期执行，那这两种机制有何不同呢？

第一，模板元编程偏向于函数式范式，而常量表达式元编程偏向于命令式范式。函数式范式通常使用不可变数据来避免状态变化，而命令式范式通常依赖于变量赋值，状态会频繁发生变化。因此，模板元编程只能使用常量，无法使用变量，而常量表达式元编程既能够使用常量，也能够使用变量。

第二，模板元编程采用特化和递归来控制流程，而常量表达式元编程采用条件和循环来控制流程。因为前者只能使用常量，而后者却允许变量赋值。

第三，模板元编程的参数可以是类型和值，而常量表达式元编程的参数只能是值。前者的参数必定产生于编译期，而后者的参数既可能产生于运行期，也可能产生于编译期。

第五，模板元编程具有强编译期保证，而常量表达式元编程并不总有强编译期保证。前者是一套独立的泛型编程体系，设计之初就只针对编译期代码，而后者是结合现有的编程体系，针对性不足。

第六，模板元编程更复杂，常量表达式元编程更容易。前者是独立的泛型编程体系，有着不同平常的一套编码逻辑，而后者是现有的编程体系，可以简单地将代码直接变为编译期代码。

模板元编程的复杂性自不必多提，前面几章已经展现得淋漓尽致。下面让我们来全面地学习常量表达式，看看 C++ 元编程的第二阶段是如何发展的，掌握另一种编译期代码表达方式的万般用法。

8.3 The constexpr Specifier

C++11 引入了 `constexpr` 说明符，可用来指定一个变量或函数的值能够在常量表达式中使用。该说明符只能用在两个地方：一是变量或变量模板的定义，二是函数或函数模板的声明（其他所有声明也必须使用同样的说明符）[1]。其他情况均不可使用，如：

```
constexpr struct point {};   // error: point is a type
void foo(constexpr int x) {} // error: not for parameters
extern constexpr int value;  // error: not a definition
constexpr void bar();        // ok, declaration
void bar() {} // error: missing required specifier
```

`constexpr` 函数一般需要定义在头文件中，为避免违反 ODR 所引起的重定义问题，该函数在首次声明的时候便会隐式 `inline`。C++17 开始，以 `constexpr`

[1] 虽然 `constexpr` 函数允许分离声明和定义，但是所有编译单元（cpp 文件）是并行编译的，为了在编译期对函数求值，一般都是将声明和定义同时写在头文件中。

修饰的静态成员变量也能够隐式 `inline`，这解决了静态成员变量需要类外定义的历史遗留问题。对一个对象声明来说，以 `constexpr` 修饰将会隐式 `const`，原因显而易见，因为该对象需要在编译期确定下来。除此之外，*namespace scope* 下的 `constexpr` 默认具有内部链接，若是想切换为外部链接，可以使用 `inline constexpr`。以下是对此的代码描述：

```
constexpr void f(); // implicitly inline
struct S {
    constexpr S() {}             // implicitly inline
    constexpr void g() {}        // implicitly inline
    static constexpr int x = 42; // OK, implicitly inline
                                 // since C++17
};

/* namespace scope */
// implicitly const, internal linkage (not inline)
constexpr S s;
// implicitly const, external linkage
inline constexpr S s;
```

元编程要表达逻辑、实现功能，`constexpr` 函数是重中之重，下面从 C++11 到 C++26 分别介绍一下其中的关键演变。

8.3.1 The constexpr Functions in C++11

C++11 `constexpr` 函数初被引入，尚显简陋，约束颇多，标准描述为：

1. it shall not be virtual;
2. its return type shall be a literal type;
3. each of its parameter types shall be a literal type;
4. its function-body shall be = `delete`, = `default`, or a compound-statement that contains only
   4.1. null statements,
   4.2. `static_assert`-declarations
   4.3. typedef declarations and alias-declarations that do not define classes or enumerations,

4.4. using-declarations,

4.5. using-directives,

4.6. and exactly one return statement;

5. every constructor call and implicit conversion used in initializing the return value shall be one of those allowed in a constant expression.

逐条翻译一下，说的其实是：

1. 不能是虚函数；
2. 返回类型只能是字面量类型（Literal Types），什么是字面量类型？字面量指的就是 C 中的常量，字面量类型能够在 Constant Expressions 中创建，一个具有 *constexpr constructor* 的类就是一个字面量类型；
3. 参数类型也只能是字面量类型；
4. 函数体只能是 `= delete`、`= default`，或是只包含以下内容的复合语句：
   4.1. 空语句，就是指一个什么都不做、只为保证语法正常的语句，如 `while (true);` 中的 “;”，
   4.2. `static_assert` 静态断言，
   4.3. 不用来定义类和枚举的 `typedef` 和 `using` 别名，
   4.4. using-dclarations，如 `using std::cout`，
   4.5. using-directives，如 `using namespace std`，
   4.6. 仅包含一条返回语句；
5. 初始化返回值时，每个构造函数调用和隐式转换也需要是 Constant Expressions 允许的操作。

因此，此时下面这些代码都无法编译：

```
struct A { A() {} }; // A is not a literal type
struct B {
    // error: function cannot be declared
    // both virtual and constexpr
    constexpr virtual int f() {
        return 0;
    }
```

```
};

// error: no return type
constexpr void f() {}
// error: return type is not a literal type
constexpr A g() { return A{}; }
constexpr int h() {
    // error: typedef defines a class.
    typedef struct { int x, y; } point;
    // error: using defines an enumeration.
    using color = enum { red, blue };
    return 0;
}
constexpr int next(int x) {
    return ++x; // error: use of increment
}
```

另外一点尤其需要注意，此时的 `constexpr` 函数只能有一条返回语句，下面两个函数的功能虽然等价，但写法不同，结果相去甚远：

```
// ok, exactly one return statement
constexpr int fibonacci(int n) {
    return (n <= 1) ? n :
        fibonacci(n - 1) + fibonacci(n - 2);
}

// error, body not just "return expr"
constexpr int fibonacci(int n) {
    if (n <= 1)
        return n;

    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

不仅如此，像下面这样简单的功能也无法支持：

```
// error, body not just "return expr"
constexpr int get_value() {
```

```
    int x = 42;
    return x;
}
```

因此，此时的 `constexpr` 函数相当严格，只能写一些最简单的编译期代码。

编译期函数调用会经历函数调用替换，先是将实参隐式转换为相应的形参类型，再以转换后的实参替换函数体中每次出现的形参，最后再把函数体中的返回表达式隐式转换为函数的返回类型，如：

```
constexpr int add(int x, int y) {
    return x + y;
}

int main() {
    constexpr int result = add(2, 3);
}
```

`add()` 的函数调用替换过程为：

1. 实参转换：将实参 2 和 3 分别隐式转换为形参 `x` 和 `y` 的类型，由于实参类型和形参类型本就相同，故无转换；
2. 形参替换：将 `add()` 中的 `x` 和 `y` 分别替换成 2 和 3；
3. 返回值转换：`x + y` 的结果是 5，将其隐式转换为 `int` 返回类型，此处亦无转换。

函数调用替换必须能够生成一个常量表达式，若不满足该条件，则程序是 IFNDR(ill-formed, no diagnostic required)，如：

```
constexpr int f(bool b) {
    // OK
    return b ? throw 0 : 0;
}
// ill-formed, no diagnostic required
constexpr int f() { throw 0; }
```

对于非静态成员函数，C++11 中以 `constexpr` 修饰会自动使该成员函数变为 `const`，这意味着无法修改任何成员变量，例如：

```
struct S {
    int x;
    constexpr S(int x) : x(x) {}
    constexpr bool update(int x) {
        // error: assignment of x in read-only object
        return (this->x = x, true);
    }
};
```

这其实会极大缩窄 `constexpr` 函数的有效范围，对象的状态无法在编译期改变。

总之，为支持编译期求值，`constexpr` 函数在 C++11 的限制极严，还属于函数式范式，并不支持状态改变，变量赋值、条件循环等最核心的命令式范式功能尚难实现。

8.3.2 The constexpr Functions in C++14

到了 C++14，`constexpr` 函数进一步发展，放宽了诸多限制，主要存在三点变化。

第一点，允许编写更多语句，支持状态改变，标准中的措辞由 C++11 的“只能包含以下内容”变成 C++14 的“不能包含以下内容”，可见限制内容从不可数变成了可数，宽松很多。具体而言，此时函数体内不能包含以下内容：

1. asm-definition，
2. `goto` 语句，
3. try-block，
4. 非字面值类型变量、静态变量、线程存储期变量、未初始化变量的定义。

如今已经可以在编译期编写一些复杂的逻辑，例如，以下是一个以动态规划算法实现的编译期计算斐波那契数列代码：

```
constexpr int fibonacci(int n) {
    int memo[n + 2] = { 0, 1 };
    for(int i = 0; i < n; ++i) {
        memo[i + 1] += memo[i];
        memo[i + 2] += memo[i];
```

```
    }
    return memo[n];
}

constexpr int result = fibonacci(10);
```

不仅可以用变量，还可以使用条件（`if` 和 `switch`）和循环（`for`、Range-based `for`、`while` 和 `do-while`），这是从函数式范式向命令式范式迈进的一大步，编译期代码再也不用只依赖于递归了。

第二点，移除了 `constexpr` 非静态成员函数默认 `const` 的规则，以允许在编译期改变对象，现在下面的代码变得合法：

```
struct S {
    int x;
    constexpr S(int x) : x(x) {}
    constexpr void update(int x) {
        // ok since C++14
        this->x = x;
    }
};
```

第三点，以下库开始支持 `constexpr`：

- `<array>`
- `<chrono>`
- `<complex>`
- `<initializer_list>`
- `<tuple>`
- `<utility>`

一个小例子：

```
constexpr std::array<int, 5> arr = { 1, 2, 3, 4, 5 };
constexpr auto duration = std::chrono::seconds(arr[0] +
↪  arr[1]);
constexpr std::complex<double> complex(arr[2], arr[3]);
```

```
constexpr auto tuple = std::make_tuple(duration.count(),
    'x');
constexpr auto pair = std::make_pair(arr[1], arr[2]);
constexpr std::initializer_list<int> list = { 1, 2 };
```

其中，`std::initializer_list<int>` 有些不同，常见的实现方式是以指针来访问其底层的临时数组，如果该数组不属于静态存储期，那么会使整个初始化也不属于一个常量表达式，产生编译失败。如：

```
// OK
constexpr std::initializer_list<int> global_list = { 1, 2 };

int main() {
    // error: constexpr variable 'local_list' must be
    // initialized by a constant expression
    // note: pointer to subobject of temporary is not
    // a constant expression
    constexpr std::initializer_list<int> local_list = { 1, 2
    };

    // OK
    static constexpr std::initializer_list<int> static_list =
    { 1, 2 };
}
```

这三点变化无疑是令 `constexpr` 函数初具雏形，真正具备一些命令式范式能力，可以用来简化一些原本只能借助模板元编程才能实现的编译期代码。

8.3.3 The constexpr Functions in C++17

C++17，`constexpr` 函数继续发展，主要存在两点变化。

第一点，支持 *constexpr lambda* 表达式。例如，下面采用 Lambda 表达式改写了上一节实现的 `fibonacci()` 函数：

```
constexpr auto fibonacci = [](int n) {
    int memo[n + 2] = { 0, 1 };
    for(int i = 0; i < n; ++i) {
```

```
        memo[i + 1] += memo[i];
        memo[i + 2] += memo[i];
    }
    return memo[n];
};

static_assert(fibonacci(10) == 55);
```

这点更新主要是增强了 `constexpr` 函数的便捷性和完整性，使得 Lambda 函数也可以像其他函数那样在编译期计算。

第二点，支持 *constexpr if* 语句。这是极具跨时代意义的一个编译期特性，能够避免模板偏特化、标签分发等传统逻辑分派方式在递归时所依赖的辅助函数。借助此特性，可以直接在当前函数内判断终止状态，不再需要单独定义一个递归终止函数，§6.7 是一个绝佳的对比示例。

constexpr if 具有两种形式，语法如下：

```
// #1
if constexpr ( condition ) statement
// #2
if constexpr ( condition ) statement else statement
```

使用时有三个注意点，一是 `#2` 语法形式中的 `else` 不可以省略，省略会使未匹配分支中的内容无法丢弃，二是非模板函数中会检查所有分支中的代码，即便是会丢弃的语句，三是不要轻易合并多个条件，应该优先采用多个嵌套的方式。例如：

```
template<typename T>
constexpr auto point_1_ok(const T& val) {
    // If the condition is true, the "false" branch is
    // discarded at compile time. Conversely, if the
    // condition is false, the "true" branch is discarded.
    if constexpr (std::is_integral_v<T>)
        return 100;
    else
        return "hello";
}

```

```
12 template<typename T>
13 constexpr auto point_1_err(const T& val) {
14     if constexpr (std::is_integral_v<T>)
15         return 100;
16
17     // This statement is not discarded even when the
18     // condition is true, which can lead to a compilation
19     // error if the types are incompatible.
20     return "hello";
21 }
22
23 constexpr auto p1_ok  = point_1_ok(10);  // Ok
24 constexpr auto p1_err = point_1_err(10); // Compile error
25
26 template<typename>
27 constexpr void point_2_ok() {
28     if constexpr (false) {
29         int i = 0;
30         int *p = i; // Ok
31     }
32 }
33
34 constexpr void point_2_err() {
35     if constexpr (false) {
36         int i = 0;
37         // Error even though in discarded statement
38         int *p = i;
39     }
40 }
41
42 template<typename T>
43 constexpr auto point_3_ok(const T& val) {
44     if constexpr (std::is_integral_v<T>) {
45         if constexpr (T{} < 10) {
46             return val * 2;
```

```
47          }
48      }
49      return val;
50  }
51
52  template<typename T>
53  constexpr auto point_3_err(const T& val) {
54      if constexpr (std::is_integral_v<T> && T{} < 10) {
55          return val * 2;
56      }
57      return val;
58  }
59
60  // Ok
61  constexpr auto p3_ok  = point_3_ok("hello");
62  // Compile error
63  constexpr auto p3_err = point_3_err("hello");
```

C++17 的这两点变化，再加上 Fold Expressions，使其成为了一个关键的版本，从根本上增强了编译期代码的可读性和简洁性。

8.3.4 The constexpr Functions in C++20

C++20 引入了更多 `constexpr` 函数相关的新特性，由此常量表达式元编程的羽翼渐丰。

首先，`constexpr` 的约束再一次降低，函数体内允许包含 `try-block` 和内联汇编，也开始允许声明 `constexpr virtual` 函数和 `constexpr` 析构函数。如：

```
1  struct S {
2      constexpr S() = default;
3      // OK since C++20
4      constexpr ~S() = default;
5
6      // OK since C++20
7      constexpr virtual int f(int x) {
8          try { return x + 1; }
9          catch(...) { return 0; }
```

```
    }
};
```

不过，即使是允许 `try-block` 和内联汇编，在常量表达式中也仍然禁止抛出异常或是执行内联汇编。

其次，`std::string` 和 `std::vector` 也允许在常量表达式中使用，其成员函数纷纷支持 `constexpr`。然而，因涉及动态内存分配，它们只能在 `constexpr` 或 `consteval` 函数内使用，如何解决这个问题呢？§8.8 有详尽的讨论。

最后，以下库、函数或类也开始支持 `constexpr`：

- `<algorithm>`
- `<utility>`
- `<complex>`（追加更多编译期操作）
- `std::swap()`（及相关函数）
- `std::array`（追加编译期比较操作）
- `std::pointer_traits`
- Misc constexpr bits
- `std::allocator`（及相关工具）
- `std::invoke()`（及相关工具）
- `std::atomic` 和 `std::atomic_flag` 的默认构造函数
- numeric algorithms

一个小例子：

```
constexpr auto f() {
    std::array<char, 6> a{ 'H', 'e', 'l', 'l', 'o' };
    auto it = std::find(a.begin(), a.end(), 'o');

    std::vector<int> vec(10);
    std::iota(vec.begin(), vec.end(), 1);

    return std::invoke([](int val) { return val + 10; },
        vec[1]);
}
```

8.3.5 The constexpr Functions in C++23

C++23 继续发展了 `constexpr` 函数，主要存在五点变化。

第一点，允许在 `constexpr` 函数内使用非字面量变量、标签和 `goto`。例如:

```
template<typename T> constexpr bool f() {
    if (std::is_constant_evaluated()) {
        return true;
    } else {
        T t;
        return true;
    }
}
struct nonliteral { nonliteral(); };
static_assert(f<nonliteral>());
```

第二点，允许在 `constexpr` 函数内定义 `static` 和 `thread_local` 变量。如:

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

C++23 以前，若要达到同样的目的，需要借助一点技巧，通过下面这个辅助函数来实现：

```
template<auto Data>
constexpr const auto& make_it_static() {
    return Data;
}
```

这个技巧的具体使用细节，见 §8.7。

第三点，去除 `constexpr` 函数参数和返回值必须是字面量类型的限制。

第四点，`constexpr if` 条件可以从整型转换为布尔类型，见 Table 8.1。

第五点，以下库、函数或类开始支持 `constexpr`:

- `<cmath>`
- `<cstdlib>`
- `std::unique_ptr`

Table 8.1: Narrowing contextual conversions to bool in `if constexpr`

Before	After
`if constexpr(bool(flags & Flags::Exec))`	`if constexpr(flags & Flags::Exe」c)`
`if constexpr(flags & Flags::Exec != 0)`	`if constexpr(flags & Flags::Exe」c)`

- `std::to_chars()` 和 `std::from_chars()`（整型重载）
- `std::bitset`

其中，`constexpr std::unique_ptr` 使得可以在编译期动态分配内存，下面的用法成为可能：

```
constexpr std::unique_ptr<int[]> create_array(int size) {
    auto arr = std::make_unique<int[]>(size);
    for (int i = 0; i < size; ++i) {
        arr[i] = i * i;
    }
    return arr;
}

int main() {
    constexpr int size = 5;
    auto arr = create_array(size);

    for (int i = 0; i < size; ++i) {
        std::cout << arr[i] << " "; // Output: 0 1 4 9 16
    }
}
```

`constexpr std::to_chars/std::from_chars` 则允许在编译期实现字符串与数值之间的转换，是性能最高的转换方式[2]，下面是一个示例：

```
constexpr std::optional<int> to_int(std::string_view s) {
    int value;
    if (auto [p, err] = std::from_chars(s.data(),
```

[2]该主题的详细讨论见：https://www.cppmore.com/2023/03/28/comprehensive-c-string-to-int-conversion-benchmarksc89-c23/

```
        s.data() + s.size(), value); err == std::errc{}) {
        return value;
    } else {
        return std::nullopt;
    }
}

int main () {
    constexpr std::string_view sv = "20230317";
    constexpr int result = *to_int(sv);
    static_assert(result == 20230317);
}
```

8.3.6 The constexpr Functions in C++26

截止目前[3]，C++26 也带来了五点有关 `constexpr` 函数的更新。

第一点，允许编译期 `void*` 转换，以支持 `constexpr` 类型擦除，P2738R1[4] 中有一个不错的例子：

```
struct Sheep {
    constexpr std::string_view speak() const noexcept {
        return "Baaaaaa";
    }
};

struct Cow {
    constexpr std::string_view speak() const noexcept {
        return "Mooo";
    }
};

class Animal_View {
private:
    const void *animal;
```

[3] 2024 年 11 月

[4] https://wg21.link/p2738r1

```
    std::string_view (*speak_function)(const void *);
public:
    template <typename Animal>
    constexpr Animal_View(const Animal &a)
    : animal{&a}, speak_function{[](const void *object) {
        return static_cast<const Animal *>(object)->speak();
    }} {}

    constexpr std::string_view speak() const noexcept {
        return speak_function(animal);
    }
};

// This is the key bit here. This is a single concrete
// function that can take anything that happens to have
// the "Animal_View" interface
std::string_view do_speak(Animal_View av) { return
↪   av.speak(); }

int main() {
    // A Cow is a cow.
    // The only think that makes it special is that
    // it has a "std::string_view speak() const" member
    constexpr Cow cow;
    // cannot be constexpr because of static_cast
    [[maybe_unused]] auto result = do_speak(cow);
    return static_cast<int>(result.size());
}
```

不同于 Subtype Polymorphism，类型擦除所实现的多态不需要虚表，一旦允许 `constexpr` 转换 `void*`，便可实现编译期的类型擦除式多态。基于此特性，`std::format`、`std::function_ref`、`std::function` 和 `std::any` 等组件也可以在编译期工作。

第二点，允许在编译期评估 `placement new`，定义已改为：

```
namespace std {
```

```
    // [new.delete], storage allocation and deallocation
    [[nodiscard]] constexpr void* operator new  (
        std::size_t size, void* ptr) noexcept;
    [[nodiscard]] constexpr void* operator new[](
        std::size_t size, void* ptr) noexcept;
}
```

第三点，std::stable_sort、std::stable_partition 和 std::inplace_merge 支持在常量表达式中使用。

第四点，进一步使 <cmath> 和 complex 中的更多操作支持编译期，如pow、log、sqrt 等。

第五点，允许在 constexpr 代码中抛出异常:

```
consteval auto hello(std::string_view input) {
    if (input.empty()) {
        throw invalid_argument{"empty name provided"};
    }

    return concat_into_a_fixed_string("hello ",input);
}

// compile-time error at call site "empty name provided"
const auto a = hello("");
const auto b = hello("Hana");

try {
    const auto c = hello("");
} catch (const validation_error &) {
    // everything is fine
}
```

这个提案由 Hana Dusíková 提出，编译期异常其实是没有运行期异常所附带的一些缺点的，静态反射就打算采用这种方式作为首要的错误处理方式，在第九章中还会单独讨论这一点。

以上这些更新，再次促进了编译期和运行期编程的统一。constexpr 函数既能够在编译期执行，也能够在运行期执行，而 C++20 出现了两个新的说明符，可以强制变量或函数在编译期执行，下面便来看看其异同。

8.4 The consteval and constinit Specifiers

C++20 引入了两个新的说明符：一个是 `consteval`，另一个是 `constinit`。`consteval` 用来指定一个函数是 *immediate function* ——其调用必须产生一个编译期常量，只能用在函数或函数模板的声明；`constinit` 用来断言一个变量拥有静态初始化（即 *zero initialization* 和 *constant initialization*），否则程序是非良构的。

与 `constexpr` 相同，以 `consteval` 修饰的函数也会隐式 `inline`，不过，析构函数、内存分配函数和内存释放函数不能声明为 `consteval`，如：

```
struct S {
    constexpr S() {};
    consteval ~S() {};

    inline static int buffer[1024] = {};
    consteval static void* operator new(std::size_t size) {
        // return something, exposition only
        return &buffer;
    }

    consteval static void operator delete(void* p) {}
};
```

全局的 `operator new` 即便是 `constexpr` 也无法重载，因为一旦定义了重载，先前的所有声明也必须要包含 `constexpr` 说明符。

`consteval` 函数与 `constexpr` 函数之间最大的不同在于前者会强制保证在编译期调用，而后者没有这种保证。这种强制保证编译期执行的调用方式称为 *immediate invocation*，这种调用必须要产生一个常量表达式，约束更严。例如：

```
consteval int next(int n) {
    return ++n;
}

consteval int next_next(int n) {
    return next(next(n)); // OK
}

```

```
constexpr int double_next(int n) {
    return next(next(n)); // Error
}
```

因此，在需要强保证执行编译期的情境下，应该优先考虑 `consteval`，后续标准增加的元函数就清一色地以 `consteval` 修饰。

`constinit` 则会强制保证变量处于编译期，如：

```
const char* g() { return "dynamic initialization"; }
constexpr const char* f(bool p) {
    return p ? "constant initializer" : g(); }

constinit const char* c = f(true);      // OK
constinit const char* d = f(false);     // error
```

`constinit` 相当于只是一个编译期的变量声明断言，如果变量没有在编译期初始化，就产生编译错误，它并不会隐式添加 `const` 和 `inline`。`constinit const` 其实与 `constexpr` 等价，没有必要这么写，而 `inline constinit` 在大多数情况下也没有什么必要，对全局作用域下的 `constinit static` 变量加上 `inline` 修饰是多此一举，不如直接删去 `static`。但是，对局部作用域下的 `constinit static` 变量加上 `inline` 却可以避免类外定义静态变量，如：

```
struct S {
    inline constinit static int x = 5; // ok
    constinit static int y = 5;        // error
};
```

我在 *The Book of Modern C++*[5] Chapter 3 对此有详细讨论。

8.5 Constant Allocations

动态内存分配是常量表达式中的一大难点，不解决这个问题，便犹如弓缺强弦，非静态标准容器都无法在编译期使用。编译期的动态内存分配并不像乍想之下那般简单，要不也不会迟迟没有完全解决。细分下来，其实存在三种类型的编译期分配：Transient Allocation（瞬态分配）、Non-Transient Allocation（非瞬态分配）和 Persistent Allocation（持久分配），C++ 目前只是支持第一种而已，限制实多。

[5] https://github.com/lkimuk/the-book-of-modern-cpp

下面就来详细介绍一下各种编译期分配类型的秘奥精微。

8.5.1 Transient Allocation

C++20，P0784R7[6] 带来了 Transient Allocation，初步支持了编译期容器操作，其生命周期仅在它所处的常量求值阶段，一旦离开该阶段，便会释放内存。C++20 的 `constexpr std::string/std::vector` 以及 C++23 的 `constexpr std::unique_ptr` 都属此列，它们只能用在所处的常量求值阶段，如：

```
consteval auto f() {
    constexpr std::string str = "test"; // #1 error
    return str.size();
}

consteval auto g() {
    std::string str = "test"; // ok
    return str.size();
}

constexpr std::string str = "test";             // #2 error
constexpr auto n = std::string("test").size(); // ok
```

编译通过的是 Transient Allocation，失败的两个颇为细微，`#1` 是 Non-Transient Allocation 所要支持的，而 `#2` 是 Persistent Allocation 所要支持的，后续两节再来细论。

可见，Transient Allocation 的作用范围极窄，非但不能跨到运行期，亦不能覆盖整个编译期。究其原因，尚有两个主要问题未曾解决，一是常量销毁问题，二是常量访问问题。Barry Revzin 在 P2670R0[7] 提供了一个例子：

```
// simplified version of unique_ptr
template<class T>
class unique_ptr {
    T* ptr;
public:
    explicit constexpr unique_ptr(T* p) : ptr(p) { }
```

[6] https://wg21.link/p0784r7
[7] https://wg21.link/p2670r0

```
7       constexpr ~unique_ptr() { delete ptr; }
8
9       // unique_ptr is shallow-const
10      constexpr auto operator*() const -> T& { return *ptr; }
11      constexpr auto operator->() const -> T* { return ptr; }
12
13      constexpr void reset(T* p) {
14          delete ptr;
15          ptr = p;
16      }
17  };
18
19  constexpr unique_ptr<unique_ptr<int>> ppi(new
    ↪  unique_ptr<int>(new int(1)));
20
21  int main() {
22      // this call would be a compile error, because ppi is a
23      // const object, so we cannot call reset() on it
24      // ppi.reset(new unique_ptr<int>(new int(2)));
25
26      // but this call would compile fine
27      ppi->reset(new int(3));
28  }
```

这是一个假设编译期分配成立的例子，注意 `ppi.reset()` 和 `ppi->reset()` 的区别，前者调用的是 `ppi` 对象的成员函数，而后者调用的是 `ppi.ptr` 所指对象的成员函数。`ppi` 对象具有 `const` 限制符，`ppi.ptr` 指向的对象却没有，于是只有前者调用 `reset()` 函数时会产生编译错误。`ppi.ptr` 本身是一个只读指针（`unique_ptr<int>* const`），不可以改变，但它指向的 `ppi.ptr->ptr` 是一个可变对象（`int*`），故而在常量销毁期间可能会读取到一个可变的分配，编译期和运行期的析构行为不再一致，编译期在析构函数中所做的资源清理检查工作也变得毫无意义。我们希望的是二者都能产生编译错误，如此一来，便能够避免在运行期间释放在编译期分配的内存，保证析构行为的一致性。

既然如此，直接禁止嵌套类型不就可以避免这个问题吗？试想一下，如果不支持 `constexpr std::vector<std::vector<int>>`、`constexpr std::vector<std::string>>` 等类型，这种解决方案又能够有多大用处？未来是否还是得直面

这个问题？

事实上，尽管 `std::vector` 和 `std::unique_ptr` 都具有成员，其行为却并不一样：

```
// std::vector is deep-const,
// so its elements cannot be modified.
const std::vector<int> v = {1, 2, 3};
v[0] = 10; // error

// std::unique_ptr is shallow-const,
// we can modify the pointed object.
const std::unique_ptr<int> p(new int(5));
*p = 6; // ok
```

`std::vector` 是 Deep-const，而 `std::unique_ptr` 是 Shallow-const。Deep-const 对象的成员及其本身都是不可修改的，而 Shallow-const 对象只有其本身是不可修改的。对于 Deep-const 对象，嵌套类型会传递 `const`，因此前面所说的常量销毁问题并不存在，只有 Shallow-const 对象才存在此问题。因此，最终的设计目标并不是要禁止嵌套类型，而是要禁止 Shallow-const 嵌套类型：

```
constexpr vector<int> vi = {1, 2, 3}; // ok
constexpr vector<vector<int>> vvi =  {{1, 2}, {3 4}}; // ok
constexpr vector<string>> vs =
    { "this", "should", "work" }; // ok
constexpr unique_ptr<int> pi(new int(1)); // ok
constexpr unique_ptr<unique_ptr<int>> ppi(
    new unique_ptr<int>(new int(2))); // err
```

对于常量访问问题，可以参考 Barry Revzin 在文章[8] 中引用的 Jeff Snyder 提供的一个例子：

```
constexpr auto p = unique_ptr<int>(new int(42));

auto foo() -> void {
    array<int, *p> xs;                  // #1
    for (auto i = 0; i < *p; ++i) {     // #2
        xs[i] = i;
```

[8] https://brevzin.github.io/c++/2024/07/24/constexpr-alloc/

```
    }
}

auto main() -> int {
    *p = 47;                                // #3
    foo();
}
```

编译期分配成立的话，编译期分配的 `p` 能够在运行期使用，理所应当，也能够在运行期修改。那么 `*p` 在 `#1` 和 `#2` 的值将不一致，`#1` 处其值只能是 42，因为它是模板参数，必须在编译期确定，而 `#2` 处其值只能是 47，因为它是在运行期访问的结果。于是，上下两行看似相同的访问代码，实际上却得到了不同的值，数组在不经意间访问越界。

要解决这个问题，只能禁止 `#1` 处的常量访问，若是禁止其他两处，就相当于禁止了 Persistent Allocation，自然不行。

以上就是 Transient Allocation 不能一步到位持续到运行期的主要原因，下面继续来看 Non-Transient Allocation 的概念和意义。

8.5.2 Non-Transient Allocation

Non-Transient Allocation 的生命周期覆盖整个编译期，但无法持续到运行期。上节出现的：

```
consteval auto f() {
    // Non-transient allocation:
    // `str` exists for the duration of `f()`
    // and is destroyed at the end.
    // The allocation does not persist to runtime.
    constexpr std::string str = "test";
    return str.size();
}
```

便是一个例子。这种分配可以通过一个简单的规定来实现：常量求值期间，禁止修改对 `constexpr` 变量所作的分配。之所以简单，是因为这种分配方式不用持续到运行期，从而可以绕开前面的难题，暂时扩大一点编译期分配的能力，实现一些想要的功能。然而，这种解决方式依旧存在一致性问题，Barry Revzin 给出了一个例子：

```cpp
constexpr auto a = unique_ptr<int>(new int(1));
// must be ill-formed, *a isn't constant
static_assert(*a == 1);

constexpr auto f() -> void {
    constexpr auto b = unique_ptr<int>(new int(2));
    // must be ill-formed, *b isn't constant
    static_assert(*b == 2);
}

consteval auto g() -> void {
    constexpr auto c = unique_ptr<int>(new int(3));
    static_assert(*c == 3); // OK
}
```

此时，上下文不同，相同代码也产生了不同的行为。简单的规则导致了理解上的负担，那么这种简单还是简单吗？因此，即便是这种稍弱的编译期分配也没有直接在 C++20 或 C++23 加入，它也并不能促进 Persistent Allocation 的发展，短期内加上也没用。

Non-Transient Allocation 和 Persistent Allocation 其实是一种更加细致的分类，若仅大致分类，便只存在 Transient Allocation 和 Non-Transient Allocation，后者同时内含了 Persistent Allocation。目前的研究目标都是支持 Persistent Allocation，接着就来看看具体的概念和解决方案。

8.5.3 Persistent Allocation

Persistent Allocation 的生命周期能够从编译期一直持续到运行期，是货真价实的编译期分配。倘或支持，下面的代码都将变得合法：

```cpp
constexpr std::vector<int> f() {
    std::vector<int> v(3);
    std::ranges::iota(v, 0);
    return v;
}

constexpr auto g() {
    auto ptr = std::make_unique<int[]>(5);
```

```
9
10      for (int i = 0; i < 5; ++i) {
11          ptr[i] = i + 1;
12      }
13
14      return ptr;
15  }
16
17  int main() {
18      constexpr auto v = f();
19      constexpr auto p = g();
20      constexpr std::string str("this should be okay");
21  }
```

这才是真正有大用的编译期分配，能够在运行期继续使用编译期的常量求值结果。

对此，当前有两种解决方案，`propconst` 和 `mark_immutable_if_constexpr`。对于嵌套类型 `std::vector<std::vector<int>>` 和 `std::unique_ptr<std::uniq」ue_ptr<int>>`，前者是 Deep-const，后者是 Shallow-const，只有后者才会存在常量销毁问题和常量访问问题。因此，解决问题的思路就是区分 Deep-const 和 Shallow-const 类型，区分之后，再禁止嵌套使用 Shallow-const 类型。如何区分这两种类型呢？这便是这些解决方案所要设计的。

8.5.3.1 propconst

P1974R0[9] 意图在类型系统中引用一个 Deep-const 的标记 `propconst`，以区分不同类型。和其他 *cv-qualifier* 的用法类似，只是 `propconst` 的作用是传播 `c」onst`。提案中的一个例子：

```
1  struct S {
2      int propconst *ppi;
3      void f() const {
4          // The type of the expresion (ppi) here is
5          // "int const *const", as-if ppi was declared
6          // with "int const *"
7      }
```

[9] https://wg21.link/p1974r0

```
    void f() {
        // this->ppi's type here is "int*"
        // The type of the expresion (ppi) here is "int *",
        // as-if ppi was declared with "int*"
    }
};
```

基于此提案，Shallow-const 类型不必作任何变动，只需稍微改变一下 Deep-const 类型。比如，只需把 `std::vector` 的成员类型由 `T*` 改成 `T propconst*`。

此法的问题在于：引入一个全新的 *cv-qualifier* 影响过多，略显复杂。它将导致包括语言层面、常见容器类型、智能指针类型、类型萃取工具、前向兼容性在内的诸多变动，而这些变动已经远远脱离了原始的编译期分配问题。对此，另一种方式是尝试缩小影响范围，将 `propconst` 变成和 `mutable` 一样的 *specifier*。*cv-qualifier* 能够出现在类型出现的任何上下文中，而 *specifier* 能够出现的上下文要少一些，可以减少引入的语言规则。

总之，这种方式类似于一种主动的强制检查机制，利用这种机制可以在编译期检查类型的分配是否可以安全地持续到运行期使用，全面性检查的复杂性就是所要付出的代价。

8.5.3.2 mark_immutable_if_constexpr

相对地，这种方式类似于一种被动的无条件信任机制，它不再去追踪检查类型的所有成员，而是提供一个特殊的库函数，库作者以其标记分配为不可变状态，直接相信库作者正确地管理了分配行为。这个库函数最早出现在 P0784R5[10]，叫 `std::mark_immutable_if_constexpr()`，只是当时提案中对涉及问题和实现细节的认识尚显浅薄，后续版本中移除了对于 Non-Transient Allocation 的处理。

来看同样来自 Barry 的一个例子：

```
template <typename T>
class unique_ptr {
    T* ptr = nullptr;

public:
    constexpr unique_ptr(T* p)
        : ptr(p)
    {
```

[10] https://wg21.link/p0784r5

```
+       if constexpr (std::is_const_v<T>) {
+           std::mark_immutable_if_constexpr(ptr);
+       }
    }
};

template <typename T>
class vector {
    T* begin_;
    T* end_;
    T* capacity_;

public:
    constexpr vector(std::initializer_list<T> elems) {
        begin_ = std::allocator<T>{}.allocate(elems.size());
        end_ = capacity_ = std::uninitialized_copy(
            elems.begin(), elems.end(), begin_);
+       std::mark_immutable_if_constexpr(begin_);
    }
};
```

只要将分配使用 `std::mark_immutable_if_constexpr()` 标记一下，就可以使其持续到运行期。只要状态不可变，自然就不会存在常量销毁问题和常量访问问题。至于状态是否真的不可变，则需要库作者提供保证。此法只保证，对于标记的分配，常量求值期间的任何修改都是非法的。示例中，`std::unique_ptr` 是 Shallow-const，因此只在其成员为 `const` 才进行标记，`std::vector` 是 Deep-const，因此任何时候都可以标记。

此法的优势是模型简单，易于理解，几乎不必在编译期进行任何检查，就可以确保分配是是持久的。

至于最终哪种方式能够被标准采纳，目前尚不可知，两种方法各有利弊，否则也不会争论了这么多年。也许，C++26 只会支持 `std::vector` 和 `std::string` 的持久化分配，是直接支持，具体如何实现不必向外透露，毕竟这两个类型是编译期分配中最常用的，它们本身都是 Deep-const，不会存在问题。

关于编译期分配的讨论至此告一段落，接着，让我们继续来看常量表达式的其他内容。

8.6 Conditional and Unconditional Compile-Time Expressions

编译期编程，经常需要将函数参数当作 NTTP 与模板组合起来使用，然而，`constexpr` 函数和 `consteval` 函数并不能直接支持，看下面的例子：

```
struct S {
    int val;
    constexpr int size() const {
        return val * (val + 1) / 2;
    }
};

constexpr auto f(S s) {
    // doesn't compile
    return std::array<int, s.size()>{};
}

constexpr S s{42};
constexpr auto arr = f(s);
```

之所以编译失败，是因为函数 `f()` 不确定传递进来的参数 `s` 本身是否为编译期常量表达式（即便是 `consteval` 函数也是如此），于是在设计上便对这种情况进行了限制。反之，如果参数是通过模板参数传递的，那绝不置疑，参数肯定是编译期常量。

此时，模板参数就称为 Unconditional Compile-Time Expression，而参数 `s` 则称为 Conditional Compile-Time Expression，后者必须转换为前者，才能对参数提供强编译期保证。

其实，早在 2019 年，P1045R1[11] 就提议过 `constexpr` 函数参数，在该提案中，`void f(constexpr int x)` 与 `template<int x> void f()` 属于等价写法。如此一来，需要模板参数的各类组件便可以进一步简化调用语法，如：

```
// Now
auto t = std::tuple<int, std::string>{};
std::get<0>(t) = 1;
```

[11] https://wg21.link/p1045r1

```
std::get<1>(t) = "asdf";
std::get<2>(t) = 3; // compile error

static_assert(
    pow<3>(2_meters) == 8_cubic_meters
);

// P1045
auto t = std::tuple<int, std::string>{};
t[0] = 1;
t[1] = "asdf";
t[2] = 3; // compile failure

static_assert(
    pow(2_meters, 3) == 8_cubic_meters
);
```

本节开头的例子也可以直接以这个特性轻易支持，但该提案似乎一直没有动静，当前同类场景的提案 P2781R4[12] 提出了一个 `std::constant_wrapper`，能在一定程度上解决此问题，只是没有 P1045R1 这种原生支持的方式直接易用，此提案仍在探索阶段。

接下来，让我们来看看有哪些办法能够将一个普通的函数参数转换成一个 Unconditional Compile-Time Expression 参数，提供强编译期常量表达式保证。

8.7 Two Ways to Enforce Compile-Time Guarantees

第一种方式，借助 NTTP，将参数以模板参数传递。代码如下：

```
template<auto> struct compile_time_param {};
template<auto Data> inline auto compile_time_cast =
↪   compile_time_param<Data>{};

template<S s>
consteval auto f(compile_time_param<s>) {
    // ok
```

[12] https://wg21.link//p2781r4

```
    return std::array<int, s.size()>{};
}

constexpr S s{ 42 };
constexpr auto arr = f(compile_time_cast<s>);
```

第二种方式，可以传递一个 Lambda 作为 `constexpr` 函数的参数，通过执行该 Lambda 来得到一个编译期常量表达式。代码如下：

```
template<typename Callable>
consteval auto f(Callable callable) {
    // ok
    return std::array<int, callable().size()>{};
}

constexpr S s{ 42 };
constexpr auto make_compile_time_data = [] { return s; };
constexpr auto arr = f(make_compile_time_data);
```

这两个技巧在编译期编程中大有用武之地，一定要掌握。

8.8 Persisting Compile-Time Strings for Runtime Use in C++20

因为目前只支持 Transient Allocation，所以 `constexpr std::string` 用起来格外受限。如：

```
constexpr std::string make_string() {
    std::string str{"compile time string"};
    return str;
}

int main() {
    // error
    constexpr std::string s1 = "compile time string";
    // error
    constexpr std::string s2 = make_string();
```

```
    // ok
    static_assert(make_string() == "compile time string");
}
```

Transient Allocation 的生命周期仅在常量求值阶段，这意味着我们只能将 `std::string` 放在 `constexpr/consteval` 修饰的函数里面使用，也因此，保存 `constexpr std::string` 的值也成了一种奢望。此时，就需要一些手段来达到目的，总的来说，有两种解决思路。

第一种思路，在 `constexpr std::string` 的生命周期结束之前，将结果保存到一个持久性的容器里面。`constexpr std::array` 就是这样的一种容器，它的大小在编译期便已确定，不涉及动态内存分配，可以继续在运行期使用。具体实现方式如下：

```
constexpr auto make_string(int n) {
    std::string str;
    std::array<char, 10> buf;
    for (auto i : std::views::iota(0, n)) {
        // Convert integers to strings with
        // constexpr to_chars in C++23
        if (auto [ptr, ec] = std::to_chars(buf.data(),
            buf.data() + buf.size(), i); ec == std::errc())
            str += std::string_view(buf.data(), ptr);
    }
    return str;
}

constexpr auto get_length(std::string_view str) {
    return str.size();
}

template<std::size_t Len>
constexpr auto get_array(std::string_view str) {
    std::array<char, Len + 1> buf{ 0 };
    std::copy(str.begin(), str.end(), buf.begin());
    return buf;
}
```

```
24
25 int main() {
26     static_assert(make_string(11) == "012345678910");
27     constexpr static auto length =
28         get_length(make_string(11));
29     constexpr static auto buf =
30         get_array<length>(make_string(11));
31     constexpr static auto str =
32         std::string_view(buf.begin(), buf.size());
33     std::cout << str << "\n"; // Output: 012345678910
34 }
```

其中，make_string() 函数接受一个数值，然后将 [0, n) 的数值依次转换成字符串，再保存到 constexpr std::string 之中。整个函数都在编译期执行，于是像 std::to_string 之类的转换函数都无法使用，这里借助 C++23 的 constexpr std::to_chars 来完成这项工作。constexpr std::array 需要指定长度，使用之前必须得到数据的长度，这项工作交由 get_length() 函数完成。

其后，get_array() 函数将 constexpr std::string 的内容全部拷贝至 constexpr std::array，如此一来，这些内容的生命周期就可以延长到运行期。也因此，我们最终才能够以此构造一个 std::string_view，再通过 std::cout 来输出这个结果。

这种方式的缺点就是要构造两次数据，因为在调用 get_array() 函数之前，必须先调用一次 make_string() 来获取数据长度。

第二种思路同样是借助 constexpr std::array，但却要优化其缺陷——重复构造数据。优化手法就是先选择一个比较大的长度，得到一个不符合实际大小的 std::array：

```
1 constexpr auto get_array_helper(std::string_view str) {
2     std::array<char, 10 * 1024 * 1024> buf;
3     std::copy(str.begin(), str.end(), buf.begin());
4     return std::make_pair(buf, str.size());
5 }
```

在这个函数里面，将实际长度保存起来，这里返回值为一个 std::pair，第一个值就是过长的 std::array，第二个值则是数据实际的长度。然后，再把这个过长的 std::array 缩小到实际长度：

```
constexpr auto get_array(std::string_view str) {
    // Error!
    // structured binding declaration cannot be constexpr.
    constexpr auto pair = get_array_helper(str);
    constexpr auto buf = pair.first;
    constexpr auto size = pair.second;
    std::array<char, size> newbuf;
    std::copy(buf.begin(), std::next(buf.begin(), size),
    ↪ newbuf.begin());
    return newbuf;
}
```

需要注意，以上实现是存在问题的，因为函数参数 `str` 属于 Conditional Compile-Time Expression，无法直接在常量表达式中使用，会产生编译期错误。

此时，便需采用 §8.7 介绍的方式将 `str` 转换成 Unconditional Compile-Time Expression。比如，采用 Lambda 传递这种办法修改后的代码为：

```
template<class Callable>
constexpr auto get_array(Callable call) {
    // ok
    constexpr auto pair = get_array_helper(call());
    constexpr auto buf = pair.first;
    constexpr auto size = pair.second;
    std::array<char, size> newbuf;
    std::copy(buf.begin(), std::next(buf.begin(), size),
    ↪ newbuf.begin());
    return newbuf;
}
```

之后的操作，就是根据已知的实际大小，重新构造一份符合实际大小的新数据。于是，便无需额外获取一次大小，再来调用 `get_array()`:

```
constexpr auto make_data = [] {
    return make_string(11);
};

constexpr static auto data = get_array(make_data);
```

```
constexpr static auto str =
    std::string_view{ data.begin(), data.size() };
std::cout << str << "\n"; // Output: 012345678910
```

但是，我们依旧不能一步到位地得到一个 `std::string_view`，总是要经过一个 `get_array()`，难免麻烦。因此，让我们再进一步，把所有内容整合起来，提供一个独立的函数：

```
constexpr auto to_string_view(auto callable) {
    constexpr static auto data = get_array(callable);
    return std::string_view{data.begin(), data.size()};
}

constexpr static auto str = to_string_view(make_data);
std::cout << str << "\n";
```

现在调用起来就显得非常简明了。

只是，`constexpr` 函数中使用 `static` 变量是 C++23 才有的特性，在 C++20 如何实现这一目的呢？可以采用 §8.3.5 介绍的技巧来解决：

```
template <auto Data>
constexpr const auto& make_it_static() {
    return Data;
}

constexpr auto to_string_view(auto callable) {
    constexpr auto& data =
        make_it_static<get_array(callable)>();
    return std::string_view{ data.begin(), data.size() };
}
```

这个技巧的核心还是将普通参数传递转换为模板参数传递，模板参数在编译期推导并实例化，可以提供编译期常量保证。通过这种方式，就能够达到和 `constexpr static` 相同的效果。

8.9 Compile-Time Dispatching in C++20

本节介绍消息分发的一种编译期实现法。

编程是一门非常依赖逻辑的学科，逻辑分为形式逻辑和非形式逻辑，编程就属于形式逻辑。形式逻辑指的是用数学的方式去抽象地分析命题，它有一套严谨的标准和公理系统，对错分明；而日常生活中使用的则是非形式逻辑，它不存在标准和公理，也没有绝对的对与错。

根据哲学家大卫·休谟在《人性论》中对于观念之间连接的分类，我们能够把逻辑关系分成三大类：相似关系、因果关系、承接关系。相似关系表示两个组件结构相同，去掉其中一个组件，只会使功能不够全面，并不会影响程序；因果关系表示两个组件之间的依赖性极强，没有第一个组件，就不会有第二个组件，第二个组件依赖于第一个组件而存在；承接关系表示两个组件都是局部，只有组合起来，才能构成一个整体。

消息分发就属于因果关系，我们需要依赖 A，去执行 B，没有 A 就没有 B。同样属于因果关系的术语还有逻辑分派、模式匹配、定制点的表示方式等等，它们本质上都是在描述一类东西，只是有时候侧重点不同。

条件关系也属于因果关系的范畴，是编程中逻辑最重的关系。试想如果没有 `if else`，你还能写出多少程序？世界是复杂的，问题也是复杂的，因果关系必不可少。消息分发，或称为逻辑分派，就是一种简化条件关系表达方式的技术。它适用于存在大量因果关系的情境，此时若是使用原始的 `if else`，则无法适应动态发展的世界。

C++ 中，最典型、也非常有用的一种方式就是采用 `map`，因作为 `key`，果作为 `value`，因是标识符，果是回调函数。由于这种方式发生于运行期，所以也称为动态消息分发。本节要讲的，是 C++20 才得以实现的另外一种方式，发生于编译期的静态消息分发技术。

称为消息分发，一般是在网络通信的情境下。正常情境下，程序是顺序执行的，所以完全可以使用 if else 来实现因果逻辑，因为组件与组件之间距离较近，属于同一模块；而网络情境下，一个组件可以瞬间跳跃到距离非常远的另一个组件，这两个组件甚至不在同一台设备上，一台设备可能在上海，另一台在北京，此时如何让这两个组件进行沟通？也就是说，A 组件里面的某个函数执行条件不满足，如何简单地跳到 B、C、D、E……这些组件的某个函数中去处理？这种远距离的程序因果逻辑，通过消息分发组件能够非常丝滑地表示。

消息分发的标识符一般采用字符串表示，及至 C++20，标准开始允许 *string-literal* NTTP（具体见 §5.5.3），才得以在编译期实现一套可用的相关组件。

因此首先，我们得实现一个 `string_literal` 以在编译期使用：

```
template<std::size_t N>
struct string_literal {
    // str is a reference to an array N of constant char
```

```
    constexpr string_literal(char const (&str)[N]) {
        std::copy_n(str, N, value);
    }

    char value[N];
};
```

通过这种方式，我们定义了编译期能够使用的字符串组件，它能够直接当作模板参数使用。然后，定义分发器：

```
template<string_literal... Cs>
struct dispatcher {
    template<string_literal C>
    constexpr auto execute_if(char const* cause) const {
        return C == cause ? handler<C>(), true : false;
    }

    constexpr auto execute(char const* cause) const {
        (!execute_if<Cs>(cause) && ...);
    }
};
```

代码非常精简，分发器可以包含很多“因”，使用可变模板参数 `Cs` 进行表示。如果得到一个具体的“因”，我们需要找到对应的“果”，因此免不了遍历 `Cs`，借助 Fold Expressions，一行代码，优雅搞定。通过 `execute_if` 来查找是否存在对应的“因”，也就是对比字符串是否相等，查找到则调用相应的“果”，也就是具体的处理函数 `handler<C>()`。

接着，需要定义一个默认的因果，即如果没有定义相应的处理函数时，所调用的一个默认处理函数。

```
// default implementation
template<string_literal C>
inline constexpr auto handler =
    [] { std::cout << "default effect\n"; };
```

因为是模板参数，所以它能够处理所有的“因”。通过特化，我们能够在任何地方，定义任何因果，比如：

```
// opt-in customization points
template<> inline constexpr auto handler<"cause 1"> =
    [] { std::cout << "customization points effect 1\n"; };
template<> inline constexpr auto handler<"cause 2"> =
    [] { std::cout << "customization points effect 2\n"; };
```

默认版本和定制版本之间是相似关系，即使不提供定制版本，也不会影响程序的功能。由于特化更加特殊，所以决议时会首先考虑这些因果对。

但是，此时存在巨大的重复，我们通过宏来自动生成重复代码：

```
#define _(name) template<> inline constexpr auto
↪  handler<#name>

// opt-in customization points
_(cause 1) =
    [] { std::cout << "customization points effect 1\n"; };
_(cause 2) =
    [] { std::cout << "customization points effect 2\n"; };
```

现在定制起来就更加方便、简洁。

最后，具体使用如下：

```
int main() {
    constexpr string_literal cause_1{ "cause 1" };
    constexpr dispatcher<cause_1, "cause 2", "cause 3"> d;
    d.execute(cause_1);
    d.execute("cause 2");
    d.execute("cause 3");
}
```

相比动态消息分发，这种方式有两个巨大的优势，其一是编译期，其二是定制时可以在任何地方。动态消息分发一般需要调用 `d.add_handler(cause, effect)`，因为是成员函数，所以限制了定制地方，必须得在对象所在模块，而静态消息分发这种全局定义特化的方式，并没有这种限制。

目前其实还存在两个问题，第一是 `C == cause` 并没有相应的比较操作符，第二是 `d.execute(cause_1)` 并不能直接传递，因为 `char const*` 和 `string_literal` 毕竟不是同一种类型。可以通过添加运算符重载和隐式转换来解决：

```cpp
template<std::size_t N>
struct string_literal {
    // ...

    friend bool operator==(string_literal const& s, char
    ↪ const* cause) {
        // return std::strncmp(s.value, cause, N) == 0;
        return std::string_view(s.value) == cause;
    }

    operator char const*() const {
        return value;
    }

    // ...
};
```

`std::strncmp` 并不是一个编译期函数，而 C++17 `std::string_view` 提供有编译期的字符串比较运算符重载，是以借其实现。现在以上静态消息分发组件就能够正常使用了。

当前每次执行时都会进行一次线性查找，时间复杂度为 $O(n)$。变成 $O(1)$ 也有办法，但是传入的参数就必须也得是编译期数据，否则使用不了。以下是一种实作法：

```cpp
template<auto> struct compile_time_param {};
template<string_literal Data>
inline auto compile_time_arg = compile_time_param<Data>{};

template<string_literal... Cs>
struct dispatcher {
    // ...

    template<string_literal s>
    consteval auto execute(compile_time_param<s>) const {
        handler<s>();
    }
```

```
13
14  };
15
16  _(cause 4) =
17      [] { std::cout << "customization points effect 4\n"; };
18  _(cause 5) =
19      [] { std::cout << "customization points effect 5\n"; };
20
21  int main() {
22      constexpr string_literal cause_1{ "cause 1" };
23      constexpr dispatcher<cause_1, "cause 2", "cause 3"> d;
24      d.execute(cause_1);
25      d.execute("cause 2");
26      d.execute("cause 3");
27
28      const char cause_5[] = "cause 5";
29
30      d.execute(compile_time_arg<cause_1>);    // OK
31      d.execute(compile_time_arg<"cause 4">); // OK
32      d.execute(compile_time_arg<cause_5>);    // Error
33  }
```

这里所到的就是 §8.7 所介绍的技巧。虽然能够达到目的，但灵活性很差，因为实际的数据往往是程序运行之后才产生的，根本就不可能在编译期提前知道，所以毫无用武之地。不过，提供这么一个重载，也算是多了一种选择，$O(n)$ 既可发生于编译期，也可发生于运行期，而 $O(1)$ 接口则强制发生于编译期。

完整的代码如下：

```
1  // Compiler Explorer: https://godbolt.org/z/P3rc8MGaa
2  template<std::size_t N>
3  struct string_literal {
4      constexpr string_literal(char const (&str)[N]) {
5          std::copy_n(str, N, value);
6      }
7
8      friend constexpr bool operator==(
```

```
        string_literal const& s, char const* cause) {
        return std::string_view(s.value) == cause;
    }

    constexpr operator char const*() const {
        return value;
    }

    char value[N];
};

// default implementation
template<string_literal C>
inline constexpr auto handler =
    [] { std::cout << "default effect\n"; };

#define _(name) \
    template<> inline constexpr auto handler<#name>

// opt-in customization points
_(cause 1) =
    [] { std::cout << "customization points effect 1\n"; };
_(cause 2) =
    [] { std::cout << "customization points effect 2\n"; };

template<auto> struct compile_time_param {};
template<string_literal Data>
inline auto compile_time_arg = compile_time_param<Data>{};

template<string_literal... Cs>
struct dispatcher {
    template<string_literal C>
    constexpr auto execute_if(char const* cause) const {
        return C == cause ? handler<C>(), true : false;
    }
```

```
44
45      // compile-time and run time interface, O(n)
46      constexpr auto execute(char const* cause) const {
47          (!execute_if<Cs>(cause) && ...);
48      }
49
50      // compile-time interface, O(1)
51      template<string_literal s>
52      consteval auto execute(compile_time_param<s>) const {
53          handler<s>();
54      }
55  };
56
57  _(cause 4) = [] { /* compile time statements*/ };
58  _(cause 5) = [] { /* compile time statements*/ };
59
60  int main() {
61      constexpr string_literal cause_1{ "cause 1" };
62      constexpr dispatcher<cause_1, "cause 2", "cause 3">
    ↪ dispatch;
63      // customization points effect 1
64      dispatch.execute(cause_1);
65      // customization points effect 2
66      dispatch.execute("cause 2");
67      // default effect
68      dispatch.execute("cause 3");
69
70      constexpr string_literal cause_4{ "cause 4" };
71      const char cause_5[] = "cause 5";
72
73      dispatch.execute(compile_time_arg<cause_4>);    // OK
74      dispatch.execute(compile_time_arg<"cause 5">); // OK
75      dispatch.execute(compile_time_arg<cause_5>);    // Error
76  }
```

短短数十行代码，便实现了一个威力强大的静态消息分发组件，这算是一

个最佳实践。

8.10 Compile-Time Conversion Between Strings and Integers

本节简单展示一点字符串和整型的编译期转换实现法。

字符串转整数，可以通过下面的方式：

```
int main() {
    constexpr auto to_int = [](this auto self,
        const char* str, int h = 0) -> int {
        return *str ? self(str + 1, h * 10 + *str - '0') : h;
    };

    constexpr char s[] = "20230317";
    constexpr int result = to_int(s);
    static_assert(result == 20230317);
}
```

虽然这种方式很是简陋，没有范围检查和错误处理等等额外操作，但它发生于编译期，效率很高。此外还使用了 C++23 Recursive Lambda，若要在低版本使用，只需将 Lambda 替换为常规函数即可。

整数转字符串，可以通过下面的方式：

```
// Get the number of digits in an integer
constexpr int get_length(int value) {
    return value < 10 ? 1 : 1 + get_length(value / 10);
}

// Convert an integer to a string at compile time
template<int V>
constexpr auto to_string() {
    std::array<char, get_length(V)> result{};
    int value = V;
    int index = result.size();

```

```
    // Fill the array with digits from
    // least significant to most significant
    while (value) {
        result[--index] = value % 10 + '0';
        value /= 10;
    }

    return result;
}


int main() {
    constexpr auto str = to_string<20241221>();

    // Convert the array to a string and output with quotes
    std::cout << std::quoted(
        std::string(str.data(), str.size()));
}
```

`get_length()` 负责在编译期获取数字的长度，`to_string()` 负责在编译期完成实际转换。具体逻辑不必细讲，唯一值得注意的一点就是，`to_string()` 不能直接以函数参数的形式接受实参，因为函数参数是 Conditional Compile-Time Expressions，不是常量表达式。解决方法，要么使用 §8.7 介绍的两种强制编译期技巧，要么直接以模板参数接受实参，例子中使用的便是后者。

模板参数 `v` 是常量值，无法进行状态改变，所以在 `to_string()` 中需要使用一个局部变量来将其转变为非常量值，以在转换过程中改变变量状态。模板元编程也可以实现这个逻辑，但代码的复杂度至少得提高几个层次，能够操纵状态正是常量表达式元编程的简便之处。

这两个字符串和整数互转的实现，只为展示本章讲解知识点的运用方法，实现并不完整。如果可以，`std::from_chars/std::to_chars` 从 C++23 开始支持 `constexpr`，建议直接使用标准中的函数。

8.11 Summary

本章从 C++11 到 C++26 全面介绍了常量表达式元编程的诸多概念和原理，与模板元编程不同，这种新的元编程方式偏向于命令式范式，支持状态变化，可

以直接使用条件和循环来控制流程，从而极大降低了编写编译期代码的要求。

从 C++98 只支持少数的几个常量表达式，到 C++26 几乎可以把大多数运行期代码转移到编译期，二十多年的发展已使 C++ 产生了翻天覆地的变化。尽管如此，目前还缺少 Presistent Allocation，涉及到动态分配的常量表达式尚有不少局限，这些常量求值结果无法持续到运行期使用。

下一阶段的元编程将向何处发展？又将产生怎样的巨变？下篇将进入一个全新的时期——反射元编程——看看更强大的一种元编程工具又能带来哪些惊喜吧。

Chapter 9

Static Reflection

一路艰辛，终于来到了下篇，本部分将正式步入 C++ 元编程的第三个阶段，即 C++26 将要开启的反射元编程时期。

全新的时期，C++ 将迎来新一轮翻天覆地般的变化，对产生式元编程而言，影响可谓深远，大家以往的编程方式都会受到一定的改变。

闲话不题，让我们直接来全面地学习静态反射这个强大的新特性。

9.1 The History of C++ Reflection

大概在 2015 年左右，静态反射的语言扩展工作便已启动，最终产生了一个 TS 版本（N4766[1]），为了简化与模板元编程的集成，此时的设计采用类型的形式来表达反射信息，称为 Type-Based 反射。之后，出于种种原因，SG7 转而支持早先就已提出的 Value-Based 反射（P0425R0[2]），这种反射不再利用模板进行编译期计算，转而借助常量表达式[3]。第八章详细介绍过模板元编程和常量表达式元编程的异同，后者的确更加简洁而强大，用来更加方便，这个方向在 SG7 中也是一个共识，后期的反射一直都是采用 Value-Based 形式而设计。

Value-Based 反射布局已久，C++20-26 所增加的各种 `constexpr` 扩展（`consteval` 函数、Transient Allocation、`std::is_constant_evaluated()`、`if consteval` 等），其实都是在为其铺路。在此期间，反射提案非常散乱，2022 年出现了第一个较为综合的版本 P1240[4]，各种概念才渐渐明确。2023 年，又推出更为全面

[1] https://wg21.link/n4766

[2] 具体描述，见：https://wg21.link/p0425r0

[3] https://wg21.link/p0993r0

[4] https://wg21.link/p1240r2

的版本 P2996[5]，冲击 C++26，两年之间，历经多次修改迭代，才有了进入标准的可能。

静态反射的标准化过程，真可谓是十年磨一剑，霜刃未曾试。在此期间，因为开发者们对反射的需求迟迟难以满足，所以催生了一些用户自己实现反射的方法。

这些方法，大体可以分为两个阶段。第一个阶段是动态反射时期，主要包含侵入式动态反射和非侵入式动态反射，由于缺少类型元信息机制，这些实作手法往往需要自己保存类型信息，之后再根据保存的内容获取相关的反射信息，因此束缚不少，需要用户手动注册类型，操作繁琐，类型元信息也不够完整。第二个阶段是静态反射时期，借助 `constexpr` 特性，这些实现法把类型元信息的保存工作转移到了编译期，但是依旧需要用户手动注册类型信息。2016 年，Antony Polukhin 在 CppCon 作了一场著名的演讲，题为 *C++14 Reflections Without Macros, Markup nor External Tooling*[6]，首次介绍了一种无需手动注册类型元信息的静态反射实作法，`magic_get` 库[7]展示了这些巧妙的实现方法。后面的一些反射库，如 Cista[8]，便是基于此法而实现。不需要手动注册类型信息后，便可以轻松将类型的数据保存起来，再进行恢复，只是这种方式有一个致命缺陷——只能得到成员的值，无法得到成员的名称。2023 年，kris-jusiak 展示了一种编译期获取结构体成员名称的方法[9]，也就是 §6.8 所展示的实现，进一步增强并简化了用户自定义静态反射的能力，后续一些反射库也纷纷采用了这种方法来优化相关功能。

这两个用户自定义的反射实现阶段，涉及的核心技术也就是前两个阶段的元编程，第一个阶段中的实作法主要利用的是宏元编程，而第二个阶段中的实作法则主要利用的是宏元编程和常量表达式元编程。

总之，静态反射进入标准之前，各种反射技术已是百花齐放，受益于不断增加的 `constexpr` 特性，它们的功能也愈发强大。然而，这些到底只是 Library 级别的特性，多多少少会有各种限制，无法与即将进入标准的 Language 级别的反射特性相提并论，亦无法支持反射元编程。

---

[5] https://wg21.link/p2996
[6] https://www.youtube.com/watch?v=abdeAew3gmQ&ab_channel=CppCon
[7] https://github.com/apolukhin/magic_get
[8] https://cista.rocks/
[9] https://www.reddit.com/r/cpp/comments/18b8iv9/c20_to_tuple_with_compiletime_names/

9.2 Experimental Compiler Environment

只是纸上谈兵，社区自然没甚激情，所以 SG7 早就提供了一些基于 Reflection TS（Value-Based 版本）的实现，以在社区激起一些浪花。

总体上而言，存在三个主要的实现版本：

- **Lock3**：Lock3 在 Clang 的一个分支上编写的 P1240 实现版本[10]，不仅包含提案中提到的反射特性，还支持一些与反射高度相关的特性，如 Expansion Statements、Source Code Injection。该版本早已停止维护，所以反射部分不支持最新的一些特性和语法。2022 年我第一次写标准反射文章[11]的时候，该版本是当时最完善的实现，文章中的示例便是用这一版本编写。

  Compiler Explorer 在线版本：`https://cppx.godbolt.org/`

- **Clang**：Bloomberg 开源的一个 P2996 实现版本[12]，支持较新的一些静态反射语法特性。
  Compiler Explorer 在线版本：`https://godbolt.org/z/Wx8Yzqb6z`

- **EDG**：EDG(Edison Design Group) 基于 P2996 对最新反射提案提供的支持，EDG 就是 Daveed Vandevoorde 所在的公司，他是主要技术领导之一，该公司专门研发编译器相关的技术。EDG 是闭源的，只提供给经过授权的客户使用，通常面向商业用户和企业，开发者或团队需要向 Edison Design Group 购买许可证才能使用。
  Compiler Explorer 在线版本：`https://godbolt.org/z/beT7ao7h1`

目前来说，EDG 是最新的版本，毕竟是商业版本，支持很快，比如最新的 ^^ 语法形式在写书时就只有此一个支持，因此本文采用该版本进行示例编写。尽管如此，EDG 的反射版本并不像 Lock3 那样，支持并驾齐驱的几个元编程特性，例如 Expansion statements 和 Soure code injection，它只是在反射特性上有了更新的实现。虽然也有平替方式，但是功能上要弱化许多，而且不甚方便。

9.3 Related Authors

知识似树，发枝散叶，往往只需两三人而已。欲了解一个领域，先知悉其中的几位关键人物，由此扩散挖掘，便可以快速理解该领域 80% 以上的内容。

[10] `https://github.com/lock3/meta`
[11] *The Book of Modern C++* Chapter 35
[12] `https://github.com/bloomberg/clang-p2996`

因此，本节介绍一下 SG7 中 Reflection 相关的研究人员。

先从 Wyatt Childers 说起，他是 Lock3 Software 的软件工程师，主要就是研究实现静态反射和元编程。Lock3 版本的反射就是他们写的，我在 *The Book of Modern C++* Chapter 35 中便是使用的这一版本进行示例编写。

而 Lock3 Software 公司的创始人是 Andrew Sutton，此人就是 C++20 Concepts 提案的作者，也是 GCC Concepts 的主要开发人员。他于 2010-2013 年间曾以博士后研究员的身份加入 TAMU（Texas A&M University），而 Bjarne Stroustrup 于 2002-2014 年间在 TAMU 担任计算机科学主席教授，两人就此相识。自 2012 年 Andrew Sutton 参与并实现 Concepts 之后，他便开始重度参与静态反射和元编程的设计与实现。

Barry Revzin 则是比较活跃的一位 C++ 标准委员会成员，参与过众多标准提案，比如 C++23 `Deducing this`、`if consteval`、`Formatting Ranges`、`ranges::fold`、`views::join_with`、`views::as_rvalue` 等等。他也写过许多文章，参加过一些演讲，大家曾经肯定读过他的某些文章。

Peter Dimov 是 Boost 的活跃成员，编写并维护了许多库，例如 Assert、Bind、Describe、Lambda2、Mp11、SmartPtr、Variant2，许多库后面都进了 C++11 标准，如 `shared_ptr`、`weak_ptr`、`enable_shared_from_this`、`bind` 等等。Describe 就是 Boost 当中的一个 C++14 反射库，在 *The Book of Modern C++* Chapter 33 中也介绍过。

Faisal Vali，他也是比较早期的一位贡献者，基本一直有参与静态反射的工作。他参与的比较有名的特性应该是 C++17 CTAD 和 `constexpr` Lambda。

Daveed Vandevoorde，这位也是 C++ 的早期贡献者，90 年代初便发明了各种基于模板的编程技术，并带到了 C++。早些年 C++ 模板参数必须在尖括号之间额外写一个空格，如 `list<complex<double> >`，后来不再必须，便是他的一个小贡献。他也是 *C++ Templates - The Complete Guide* 的主要作者，谁还没读过这本书呢？

因此，欲了解 Static Reflection，主要就是围绕这几人的相关论文和演讲进行，其中又以 Andrew Sutton 和 Wyatt Childers 的论文为主要资料，其他人的论文作为进一步挖掘的补充资料。

9.4 Basic Concepts of Reflection

本节介绍反射相关的几个核心概念。

9.4.1 Reflection and Reification

反射一般包含两个部分，获取和构建，如 Figure 9.1所示。

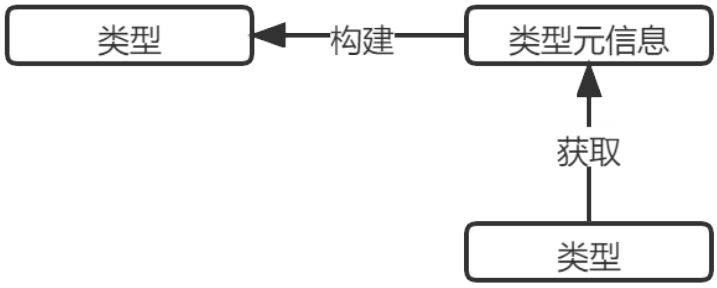


Figure 9.1: Reflection Framework

获取指的是从类型得到类型元信息，元信息要比类型高一层，所以是自下而上的结构。也就是说，这是一种从具体到抽象的结构，这个步骤就称为 **Reflection**，如 Figure 9.2所示。

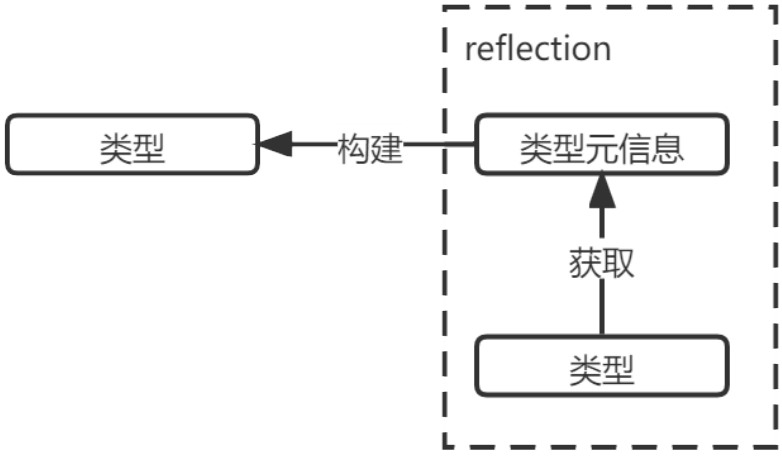


Figure 9.2: Reflection

而构建指的是从类型元信息再次得到类型，这是自上而下的结构，因此它是从抽象到具体，这个完全相反的步骤就称为 **Reification**，如 Figure 9.3所示。

这两个概念也将和后面的介绍的反射操作符关联起来。

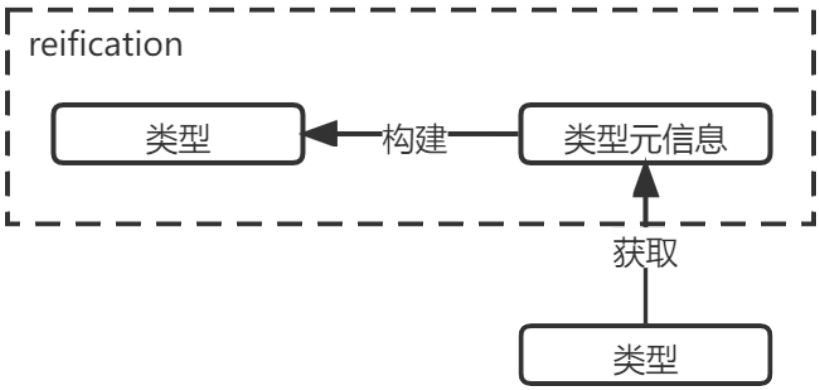


Figure 9.3: Reification

9.4.2 Introspection and Intercession

程序在运行期或编译期检查自身的结构或行为，即它能够自我观察，这种能力就称为 **Introspection**。它是反射的重要组成部分，用于获取程序的元信息（如类型、函数、属性、成员等）。

Introspection 包含许多功能，如：

- 检查类型信息：在运行期或编译期查询某个对象或变量的类型；
- 获取成员信息：枚举类的成员变量和函数；
- 检查属性和注解：提取注解或属性的元信息；
- 查询函数签名：了解函数的参数类型、返回类型、访问权限等。

这个获取元信息的能力是反射的一部分，其实 C++ 通过 Type Traits 和 Concepts 已经提供许多相关能力了，只是还不够完整。

Introspection 只涉及读取和检查程序元信息，不会对程序的结构或行为产生直接改变，与其相对的一个概念叫 **Intercession**，指的是程序能够修改或动态调整自身的结构和行为，允许程序在运行时动态地生成代码、修改类型或行为。例如，动态添加类的变量或函数、动态更改函数的实现、运行期修改类的继承关系等能力。

Introspection 和 Intercession 是反射的两大核心功能，一个用于观察（查询信息），一个用于修改（动态调整）。只有同时支持这两种能力，静态反射才能作为一种产生式元编程工具。

9.5 The Reflection(^^) Operator and Splicers([: ... :])

Lifting 和 Splicing 是静态反射的两个操作，分别对应 §9.4.1 所讲的 Reflection 和 Reification 这两个概念。

先来看一个最简单的例子：

```
struct S { unsigned i:2, j:6; };

consteval auto member_number(int n) {
    if (n == 0) return ^^S::i;
    else if (n == 1) return ^^S::j;

    std::unreachable();
}

int main() {
    S s{0, 0};
    s.[:member_number(1):] = 42; // Same as: s.j = 42;
    s.[:member_number(5):] = 0;  // Error (member_number(5)
↪   is not a constant).

    std::cout << "s.i=" << s.i << ", s.j=" << s.j << '\n';
}
```

这是一个以静态反射操纵结构体成员的小例子。

获取类型元信息的操作符是 `^^ operator`，读作 **Reflection(Lifting) Operator**，意思就是向上获取类型的元信息，对应 Reflection 这个概念。根据类型元信息重新构建出类型的语法是 `[: r :]`，称为 **Splicers**（拼接器），C++ 中把这个过程称为 **Splicing**（拼接），表示把反射的值重新具象出来，拼接到程序代码中，和 Reification 指的其实是同一个东西。

初步对应操作和概念后，接着来看它们的具体语法。

9.5.1 Reflection Operator Syntax

Reflection Operator 最初使用一个关键字 `reflexpr()` 作为语法，但这只是早期为了快速进入主要工作所采用的一个占位名称，作者们后来说这种语法 “far

too verbose”，换成了 ^，因为反射本身就有向上提取类型元信息的意思，这个新的语法形式直观而简洁，很是合适。C++ 目前存在的长关键字并不算少，如：

- `namespace`
- `decltype`
- `mutable`
- `dynamic_cast`
- `static_assert`
- `thread_local`
- ……

怎么偏偏到了 `reflexpr()` 这儿，就突然嫌繁琐了呢？一个考虑的点在于“频率”二字，Reflection Operator 太过常用，一行代码代码中可能就会出现多次，对比下面两种写法：

```
{reflexpr(int), reflexpr(double), reflexpr(char), ...}
{^(int), ^(double), ^(char), ...}
```

孰轻孰重，不言而喻。就像指针和引用一样，使用太过频繁，如果通过 `pointer` 和 `reference` 这两个关键字来代码 `*` 和 `&`，未免冗长。

只是可惜，今年作者们发现 ^ 语法与 Clang 的块扩展语法[13]起了冲突，而这一扩展在 Objective-C++ 中被广泛使用。于是，分析剩下的可用选项，深思熟虑过后，最终选择了 ^^ 作为新的语法，也就是如今使用的语法。表示为：

```
unary-expression:
    ...
    ^^ ::
    ^^ namespace-name
    ^^ type-id
    ^^ id-expression
```

^^ :: 反射的是全局命名空间，除此之外，命名空间、类型、变量、函数、模板、枚举等等也都可以作为反射的操作数。

本节只需要了解基本语法，大量实际应用例子见 §9.11。

[13] 块扩展和 Lambda 表达式相似，其语法为 `^returnType(parameters){ body }`，而 Lambda 表达式的语法是 `[capture](parameters){ body }`。

9.5.2 Splicers Syntax

Splicers 具有以下几种形式：

1. `[: r :]`
2. `typename[: r :]`
3. `template[: r :]`
4. `[:r:]::`

第 1 种不带任何修饰的形式用于产生一个表达式，第 2 种形式用来产生一个类型说明符，第 3 种形式用来产生一个模板名称，第 4 种形式用来产生一个嵌套的名称说明符。示例如下：

```
struct A { struct B {}; };
template<size_t> struct S;
constexpr auto refl = ^A;
constexpr auto tmpl = ^S;

void f() {
    // form 2)
    // Okay: declares x to be a pointer to A.
    typename[:refl:] * x;
    // form 1)
    // Error: attempt to multiply int by x.
    [:refl:] * x;
    // form 4)
    // Okay: splice as part of a nested-name-specifier.
    [:refl:]::B i;
    // form 2)
    // Okay: default-constructs an A temporary.
    typename[:refl:]{};
    // form 1)
    // Okay: operand must be a type.
    using T = [:refl:];
    // form 1)
    // Okay: base classes are types.
```

```
    struct C : [:refl:] {};
    // form 1)
    // Error: attempt to compare S with 0.
    [:tmpl:] < 0 > x;
    // form 3)
    // Okay: names the specialization.
    template[:tmpl:]<0>;
}
```

以上只为展示基本使用，区分不同语法形式，当前编译器不全支持。需要注意的是，在仅能出现类型的上下文中，`[: r :]` 其实也可以产生类型，只有在可能出现歧义的上下文中，`typename[: r :]` 才是必须的，这与 `typename` 已有的用法保持一致。

倘若将 Reflection Operator 和 Splicers 同时使用，一起一落，便相当于什么都没做。如：

```
void f() {
    typename[:^^int:] i = [:^^42:]; // Same as `int i = 42;`
    constexpr int j = 10;
    [:^^i:] = [:^^j:]; // Same as `i = j`

    // Same as `std::cout << i << '\n';`
    std::cout << [:^^i:] << '\n';
}
```

仅是示例，这种时候当然没有必要使用反射。

9.6 std::meta::info

反射得到的是类型元信息，而类型元信息的类型为 `std::meta::info`，定义为：

```
namespace std {
    namespace meta {
        using info = decltype(^::);
    }
}
```

这个类型所对应的对象就称为元对象，Reflection Operator 作用后的结果就是一个**元对象**。

针对元对象可以进行很多操作，这些操作也可以封装成各种函数，处理元对象的函数就称为**元函数**（metafunctions），而 C++ 标准事先提供的一些元函数都放在了标准元编程库中。

接着，就来瞅瞅都有哪些元函数。

9.7 The Standard Metaprogramming Library

标准元编程库，头文件为 `<meta>`，所有元函数都在 `std::meta` 命名空间下。从大的视角来看，主要包含两部分内容，一部分是元编程的通用组件，另一部分是反射操作相关的一些元函数。

9.7.1 Generalized Metafunctions

通用组件提供许多针对类型的元函数，这些元函数大多来自 `<type_traits>`，只是将那些特性加到静态反射中来，以支持元对象作为参数。比如，`type_remove_const` 所对应的声明为：

```
consteval info type_remove_const(info type);
```

换句话说，这些元函数不过是新瓶装旧酒，功能和 `<type_traits>` 中的一样，只是针对的是元对象罢了。用法也都顾名思义，例如：

```
constexpr auto MyInt =
    std::meta::type_remove_const(^^const int);
typename[:MyInt:] x = 10;
x = 42;

std::cout << x; // 42
```

这里通过 Reflection Operator 得到 `const int` 类型的元对象，再通过元函数移除该类型的 `const` 修饰，便得到了一个 `MyInt` 元对象。再通过 Splicers 将元对象重新拼接为一个类型，这代码其实就相当于 `int x = 10`。

此类元函数，不必赘述，例子中用到再提，完整列表见 https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2996r7.html#meta.synop-header-meta-synopsis。

依托反射的 Introspection 能力，本部分也有一些新的函数，比如下面的例子，检测类成员权限：

```
struct player {
public:
    float position_x;
private:
    float position_y;
protected:
    float position_z;
};

int main() {
    constexpr bool pub =
        std::meta::is_public(^^player::position_x);
    constexpr bool pri =
        std::meta::is_private(^^player::position_x);
    constexpr bool pro =
        std::meta::is_protected(^^player::position_x);

    // true false false
    std::cout << std::boolalpha
        << pub << " " << pri << " " << pro;
}
```

在静态反射之前，C++ 元编程并不具备这样的能力。上面这种返回 `bool` 值的元函数，属于 Predicates 一类，这个类别里面还有许多，列举一些如下：

```
consteval bool is_virtual(info r);
consteval bool is_override(info r);
consteval bool is_deleted(info r);
consteval bool is_defaulted(info r);
consteval bool is_user_provided(info r);
consteval bool is_user_declared(info r);
consteval bool is_explicit(info r);
consteval bool is_noexcept(info r);
consteval bool is_bit_field(info r);
```

```
consteval bool is_enumerator(info r);
consteval bool has_internal_linkage(info r);
consteval bool has_module_linkage(info r);
consteval bool has_external_linkage(info r);
consteval bool has_linkage(info r);
consteval bool is_namespace(info r);
consteval bool is_variable(info r);
consteval bool is_type(info r);
consteval bool is_type_alias(info r);
consteval bool is_namespace_alias(info r);
```

9.7.2 Reflection Metafunctions

这个部分更多偏向于反射操作相关的元函数，根据参数及返回值个数的不同，可以将它们分成单数形式和复数形式，单数形式就是参数或返回类型不是 Ranges 的元函数，而复数形式则与之相反。

先来看单数形式的元函数，列出如下：

```
// name and location
consteval auto identifier_of(info r) -> string_view;
consteval auto u8identifier_of(info r) -> u8string_view;
consteval auto display_string_of(info r) -> string_view;
consteval auto u8display_string_of(info r) -> u8string_view;
consteval auto source_location_of(info r) -> source_location;

// type queries
consteval auto type_of(info r) -> info;
consteval auto parent_of(info r) -> info;
consteval auto dealias(info r) -> info;

// object and value queries
consteval auto object_of(info r) -> info;
consteval auto value_of(info r) -> info;

// template queries
consteval auto template_of(info r) -> info;
```

```
19
20 // reflect expression results
21 template<typename T>
22 consteval auto reflect_value(T value) -> info;
23 template<typename T>
24 consteval auto reflect_object(T& value) -> info;
25 template<typename T>
26 consteval auto reflect_function(T& value) -> info;
27
28 // extract
29 template<typename T>
30 consteval auto extract(info) -> T;
31
32 // define_static_string
33 consteval auto define_static_string(string_view str)
34     -> const char *;
35 consteval auto define_static_string(u8string_view str)
36     -> const char8_t *;
37
38 // data layout
39 struct member_offsets {
40     size_t bytes;
41     size_t bits;
42     constexpr auto total_bits() const -> size_t;
43     auto operator<=>(member_offsets const&) const = default;
44 };
45
46 consteval auto offset_of(info r) -> member_offsets;
47 consteval auto size_of(info r) -> size_t;
48 consteval auto alignment_of(info r) -> size_t;
49 consteval auto bit_size_of(info r) -> size_t;
```

此类元函数数量很多，有些能够顾名思义，有些又十分常用，本节不会细讲每个的具体意义，在 §9.11 实际示例中，首次出现时会进行解释，那样理解起来会更加容易。

有一处却要说道一下，就是涉及到类型名称的元函数，它们的返回值都不

是 std::string。这是因为目前尚不支持 Persistent Allocation，constexpr std::string 的结果只能够在常量求值期间使用，不能持续到运行期，所以，如果以 std::string 来表示源代码文本，下面的例子将无法编译：

```
#include <iostream>
#include <meta>
int main() {
    int hello_world = 42;
    // Doesn't work if identifier_of produces a std::string.
    std::cout << identifier_of(^^hello_world) << "\n";
}
```

因此，目前涉及源代码文本的元函数，例如 identifier_of()，皆是以 std::string_view 或 std::u8string_view 来表示结果，倘或源字符无法表示，那么查询失败。

再来看复数形式的元函数，列出如下：

```
// template queries
consteval auto template_arguments_of(info r) -> vector<info>;

// member queries
consteval auto members_of(info r) -> vector<info>;
consteval auto bases_of(info type_class) -> vector<info>;
consteval auto static_data_members_of(info type_class)
    -> vector<info>;
consteval auto nonstatic_data_members_of(info type_class)
    -> vector<info>;
consteval auto enumerators_of(info type_enum)
    -> vector<info>;

consteval auto get_public_members(info type) -> vector<info>;
consteval auto get_public_static_data_members(info type)
    -> vector<info>;
consteval auto get_public_nonstatic_data_members(info type)
    -> vector<info>;
consteval auto get_public_bases(info type) -> vector<info>;

```

```
// substitute
template<reflection_range R = initializer_list<info>>
consteval auto can_substitute(info templ, R&& args) -> bool;
template<reflection_range R = initializer_list<info>>
consteval auto substitute(info templ, R&& args) -> info;

// reflect_invoke
template<reflection_range R = initializer_list<info>>
consteval auto reflect_invoke(info target, R&& args) -> info;
template
<
    reflection_range R1 = initializer_list<info>,
    reflection_range R2 = initializer_list<info>
>
consteval auto reflect_invoke(info target,
    R1&& tmpl_args, R2&& args) -> info;

// define_static_array
template<ranges::input_range R>
consteval auto define_static_array(R&& r)
    -> span<ranges::range_value_t<R> const>;
```

复数形式也可以称为 Range-Based 元函数，它们的返回类型为 `std::vector<std::meta::info>`，之所以不用 `std::span<std::meta::info const>`，主要是因为可以避免返回结果长久占用内存和降低复杂性。对于参数类型，可以接受任何 `type_value` 为 `std::meta::info` 的 Range，像下面这样：

```
namespace std::meta {
    template<typename R>
    concept reflection_range = ranges::input_range<R>
        && same_as<ranges::range_value_t<R>, info>;

    template<reflection_range R = initializer_list<info>>
    consteval auto substitute(info tmpl, R&& args) -> info;
}
```

复数形式中常用的元函数也很多，同样，将会在 §9.11 示例中出现时再作

细讲。

9.8 Error Handling

对于元函数中可能产生的错误，如 `template_of(^^int)`，它的参数并不是一个模板，在目前的工作草案中，处理的方法只是简单地使其无法成为一个常量表达式。

不过，P3068R6[14] 提出了编译期异常支持：

```
constexpr unsigned divide(unsigned n, unsigned d) {
    if (d == 0u) {
        throw invalid_argument{"division by zero"};
    }
    return n / d;
}

// BEFORE: compilation error due reaching a throw expression
// AFTER: still a compilation error but due the uncaught
↪  exception
constexpr auto b = divide(5, 0);

constexpr std::optional<unsigned>
checked_divide(unsigned n, unsigned d) {
    try {
        return divide(n, d);
    } catch (...) {
        return std::nullopt;
    }
}

// BEFORE: compilation error
// AFTER: std::nullopt value
constexpr auto a = checked_divide(5, 0);
```

这个特性已经进入 C++26，反射中也可以使用异常来进行错误处理。当然，

[14] https://wg21.link/p3068r6

直接 `std::expected<std::meta::info, E>` 也是一种可能的方式，但 P2996 给出了一个例子：

```
template<typename T>
requires (template_of(^^T) == ^^std::optional)
void foo();
```

如果采用 `std::expected<std::meta::info, E>` 来处理错误，将只会因比较失败而产生替换失败，比较操作依旧是常量表达式；如果采用异常，异常未被捕获时会使约束无效，比较操作不再是常量表达式，先检查一次 `T` 是否是模板，则可以使替换失败。再者，异常在运行期的诸多缺点，如性能、大小等等，在编译期都不是问题，所以未来反射会采用异常来作为错误处理机制。

9.9 Alternatives to Expansion Statements

Expansion Statements 尚未进入工作草案，而静态反射中却非常需要类似的特性，因此有了一个暂时的替代方法。

主要涉及的元函数就是 `std::meta::expand`，其实现如下：

```
namespace __impl {
    template<auto... vals>
    struct replicator_type {
        template<typename F>
        constexpr void operator>>(F body) const {
            (body.template operator()<vals>(), ...);
        }
    };

    template<auto... vals>
    replicator_type<vals...> replicator = {};
}

template<typename R>
consteval auto expand(R range) {
    std::vector<std::meta::info> args;
    for (auto r : range) {
        args.push_back(reflect_value(r));
```

```
    }
    return substitute(^^__impl::replicator, args);
}
```

实现主要是借助 `substitute` 元函数和 Fold Expressions，这个元函数的作用是用第二个参数中的反射序列来生成第一个参数中的类型，即将反射序列作为模板参数生成模板类型 `replicator`。再在 `replicator` 类中重载 `operator>>`，以 Fold Expressions 遍历反射序列，每次遍历的元素交由一个 `F` 回调处理——通常是一个 Lambda 函数。

`expand` 的参数往往是复数形式元函数的返回结果，下面是它和 Expansion Statements 用法的对照关系：

```
// Expansion Statements
template for (constexpr auto e :
    std::meta::enumerators_of(^^Enum)) {
    // handle element
}

// Expand
[:expand(std::meta::enumerators_of(^^Enum)):] >>
    []<auto e>() {
    // handle element
};
```

`replicator` 是通过模板参数接收的反射序列，所以 Expand 也需要使用模板参数来接收每个元素。

9.10 Alternatives to Source Code Injection

Source Code Injection[15] 也没有进入工作草案，而它对于产生式元编译来说至关重要，因此也存在替代方式。

主要由以下两个元函数提供支持：

```
namespace std::meta {
struct data_member_options_t {
    struct name_type {
```

[15] 第 10 章会正式讨论

```
        template <typename T>
        requires constructible_from<u8string, T>
        consteval name_type(T &&);

        template <typename T>
        requires constructible_from<string, T>
        consteval name_type(T &&);
    };

    optional<name_type> name;
    optional<int> alignment;
    optional<int> width;
    bool no_unique_address = false;
};
consteval auto data_member_spec(info type,
    data_member_options_t options = {}) -> info;
template <reflection_range R = initializer_list<info>>
consteval auto define_aggregate(info type_class, R&&)
    -> info;
} // namespace std::meta
```

`data_member_spec()` 返回给定类型数据成员声明的反射描述信息，`data_member_options_t` 用于指定数据成员的额外信息，比如名称、对齐和位域宽度，而 `define_aggregate()` 用来为一个 `class/struct/union` 声明提供实现，它的第二个参数是要往实现中注入的数据成员反射元信息序列（由 `data_member_spec()` 的返回值构成）。这就是 Source Code Injection 最基本的能力，弱化版的实现。

`define_aggregate()` 只能定义聚合类型，早先该元函数的名称一直是叫 `define_class()`，直到 P2996R8 才更改为当前这个更具体的名称。

下面来看个例子：

```
union U;
static_assert(std::meta::is_type(
    std::meta::define_aggregate(^^U, {
    std::meta::data_member_spec(^^int),
    std::meta::data_member_spec(^^char),
    std::meta::data_member_spec(^^double),
```

```
}));

// U is now defined to the equivalent of
// union U {
//   int _0;
//   char _1;
//   double _2;
// };
```

这里借助 `define_aggregate()` 和 `data_member_spec()` 元函数为类型 `U` 注入了三个成员，因为没有指定数据成员的额外信息，它们的名称默认依序生成为 `_0`, `_1`, `_2`。

需要注意，写文时，P2996R8 尚未发布，也无编译器能够支持以上代码，Clang 的实现支持 `define_aggregate`，但不支持 `^^`，EDG 则与之相反。

如果想对生成的数据成员进行更多控制，则需要借助 `data_member_option」s_t`，例如：

```
template<typename T> struct S;
constexpr auto U = define_aggregate(^^S<int>, {
    data_member_spec(^^int, {.name="i", .alignment=64}),
    data_member_spec(^^int, {.name=u8" 你好"})
});

// S<int> is now defined to the equivalent of
// template<> struct S<int> {
//   alignas(64) int i;
//               int 你好;
// };
```

此处指定了成员的名称和对齐长度，因此生成的代码中也含有这些额外的信息。`data_member_options_t` 中的内部类 `name_type` 利用 Concepts 提供了可隐式转换为 `u8string` 和 `string` 的构造，所以可以支持多种字符串类型。

`define_aggregate()` 注入的类必须是 Incomplete 类型，即不能已经存在定义，下面的用法是错误的：

```
class A {};
// Error: 'A' is already a complete type
```

```
const auto U = define_aggregate(^^A, {
    data_member_spec(^^char)
});
```

代码注入是反射元编译最重要的功能，因此这两个元函数在后面的示例中会经常看到。

9.11 Use Cases

本节展示各种使用示例，大多都是提案中已有的，也有部分是额外增加的。

9.11.1 Iterate and Print Class Members

这个例子用于展示迭代类成员的方式，代码如下：

```
struct player {
    static int x;
    double y;
private:
    float z;
protected:
    void foo() {}
};

template<typename T>
void print_members() {
    [:std::meta::expand(std::meta::members_of(^^T)):]
        >> []<auto m>() {
        std::cout << std::meta::identifier_of(m) << "\n";
    };
}

int main() {
    print_members<player>();
}
```

这个例子当前只在 EDG 上面可以编译，其他实现还不支持 `std::meta::expand`，`std::meta::identifier_of` 用来输出标识符名称。

执行结果为：

```
y
z
foo
operator=
operator=
x
```

由于其他名称输出形式暂时没有达到一个统一的标准，现在只能简单地输出一个标识符名称，对变量来说不是问题，但对函数输出来讲并不友好。也没有类似 Lock3 提供的 `__reflect_pretty_print(m)` 这种编译期打印工具，这的确是目前的一个小缺陷。

输出中的两个 `operator=` 实际上是默认生成的拷贝构造和移动构造函数的赋值运算符重载。

9.11.2 Enum to String

最经典的例子，相当于反射界的 “Hello world”。

若是所有相关特性皆支持，那么最简洁的实现如下：

```
template <typename E>
    requires std::is_enum_v<E>
constexpr std::string enum_to_string(E value) {
    template for (constexpr auto e :
        std::meta::members_of(^^E)) {
        if (value == [:e:]) {
            return std::string(std::meta::name_of(e));
        }
    }

    return "<unnamed>";
}

enum color { red, green, blue };
static_assert(enum_to_string(color::red) == "red");
static_assert(enum_to_string(color(42)) == "<unnamed>");
```

这是枚举到字符串，还有相应的反操作——字符串到枚举——实现如下：

```
template <typename E>
    requires std::is_enum_v<E>
constexpr std::optional<E> string_to_enum(
    std::string_view name) {
    template for (constexpr auto e :
        std::meta::members_of(^^E)) {
        if (name == std::meta::name_of(e)) {
            return [:e:];
        }
    }

    return std::nullopt;
}
```

由于没有 Expansion Statements 和 `std::meta::name_of`，EDG 目前支持的实现如下：

```
template<typename E>
    requires std::is_enum_v<E>
constexpr std::string enum_to_string(E value) {
    std::string result = "<unnamed>";
    [:expand(std::meta::enumerators_of(^^E)):] >>
        [&]<auto e> {
        if (value == [:e:]) {
            result = std::meta::identifier_of(e);
        }
    };
    return result;
}
```

虽说语法不太直观，倒也勉强能用。

9.11.3 List of Types to List of Sizes

类型列表转换为类型大小列表：

```
constexpr std::array types = { ^^int, ^^float, ^^double };

// the consteval is required here because
// consteval propagation (P2564) is not yet implemented
constexpr std::array sizes = []() consteval {
    std::array<std::size_t, types.size()> r;
    std::transform(types.begin(), types.end(),
                   r.begin(), std::meta::size_of);
    return r;
}();

static_assert(sizes[0] == sizeof(int));
static_assert(sizes[1] == sizeof(float));
static_assert(sizes[2] == sizeof(double));
```

这是一个相对简单的例子，只用到了 `std::meta::size_of` 这一个元函数，其作用就是得到元对象的实际类型大小。最终 sizes 的内容就相当于：

```
std::array<std::size_t, 3> sizes =
    { sizeof(int), sizeof(float), sizeof(double) };
```

9.11.4 Implementing make_integer_sequence

§5.6 展示了各种实现 `make_integer_sequence` 的方法，反射可以简化那些实现。代码为：

```
template<typename T>
consteval std::meta::info make_integer_seq_refl(T N) {
    std::vector args{^^T};
    for (T k = 0; k < N; ++k) {
        args.push_back(std::meta::reflect_value(k));
    }
    return std::meta::substitute(
        ^^std::integer_sequence, args);
}

template<typename T, T N>
```

```
using make_integer_sequence =
    [:make_integer_seq_refl<T>(N):];

static_assert(std::same_as<
    make_integer_sequence<int, 10>,
    std::make_integer_sequence<int, 10>
    >);
```

此处的核心是 std::meta::reflect_value 和 std::meta::substitute 这两个元函数，它们常搭配起来指定模板参数，构建模板特化。

std::meta::reflect_value 用于生成值的反射，以使这些值能够传递到元函数使用。其实早前还有一个复数形式的 std::meta::reflect_values，可以直接生成一批值的反射，如：

```
constexpr std::vector<int> v{ 1, 2, 3 };
constexpr std::span<std::meta::info> rv = reflect_values(v);
```

随后，便可以将这个反射列表重新拼接出来，直接作为模板参数使用，不再需要循环，如：

```
std::integer_sequence<int, ...[:rv:]...> is123;
// same as std::integer_sequence<int, 1, 2, 3>
```

只是，反射工作草案中似乎并没有包含这个元函数，以上仅是示例，也没有编译器支持。因此，只能借助 std::meta::reflect_value 搭配循环来一个一个生成反射值。

std::meta::substitute 的作用是根据已有类型，提供参数，生成新的类型。穿插一个例子：

```
template<typename ... Ts> struct X {};
template<> struct X<int, int> {};

consteval auto f() {
    auto type = ^X<int, int, float>;
    auto tmpl = std::meta::template_of(type);
    auto args = std::meta::template_arguments_of(type);
    auto new_type = std::meta::substitute(tmpl,
        std::vector(args.begin(), args.begin() + 2));
```

```
    return new_type;
}

int main() {
    static_assert(std::same_as<
        typename[:f():],
        X<int, int>>);
}
```

这个例子做的事就是通过 `std::meta::substitute` 将 `X<int, int, float>` 的最后一个模板参数移除。`std::meta::template_of` 元函数用于获取模板元对象，`std::meta::template_arguments_of` 元函数用于获取模板参数，它返回的是一个 `std::vector<info>`，这意味着必须得在常量求值期间使用，因为 `constexpr std::vector` 涉及 Transient Allocation。

所以，在本节的原始示例中，`substitute(^^std::integer_sequence, args)` 的作用就是以 `args` 中的反射序列生成一个 `std::integer_sequence` 类型。

通过这一个小例子，我们已经学习许多元函数的用法了，下面再来看其他例子。

9.11.5 Getting Class Layout

使用反射来获取类的布局信息：

```
struct member_descriptor {
    std::size_t offset;
    std::size_t size;
    bool operator==(member_descriptor const&) const =
    ↪ default;
};

// returns std::array<member_descriptor, N>
template<typename S>
consteval auto get_layout() {
    constexpr size_t N = [] consteval {
        return nonstatic_data_members_of(^^S).size();
    }();

```

```
    std::array<member_descriptor, N> layout;
    [:expand(nonstatic_data_members_of(^^S)):] >>
        [&, i=0]<auto e>() mutable {
        layout[i] =
            { .offset=offset_of(e), .size=size_of(e) };
        ++i;
    };

    return layout;
}

struct X {
    char a;
    int b;
    double c;
};

constexpr auto Xd = get_layout<X>();
static_assert(Xd.size() == 3);
static_assert(
    Xd[0] == member_descriptor{ .offset=0, .size=1 });
static_assert(
    Xd[1] == member_descriptor{ .offset=4, .size=4 });
static_assert(
    Xd[2] == member_descriptor{ .offset=8, .size=8 });
```

核心逻辑在 `get_layout()`，它用来获取一个类型的非静态数据成员信息，将信息保存在 `member_descriptor` 中。

`nonstatic_data_members_of` 元函数的作用是获取所有非静态数据成员的反射信息，它依旧返回的是 `std::vector`，因此可以看到，示例中通过 Lambda 额外提供了一次调用来获取结果的大小。等以后支持 Persistent Allocation，这额外的一次调用便可以省去。

9.11.6 A Simple Tuple Type

与传递递归继承实现法相比，一种更简单的基于反射的 Tuple 实现法：

```
/// Clang: https://godbolt.org/z/3MaGP6zqc
// Tuple definition with reflection
template<typename... Ts> struct Tuple {
    struct storage;

    // Inject members to storage with define_aggregate
    static_assert(is_type(define_aggregate(^^storage, {
        data_member_spec(^^Ts)... })));
    storage data;

    Tuple(): data{} {}
    Tuple(Ts const& ...vs): data{ vs... } {}
};

// Specialization for std::tuple_size
template<typename... Ts>
struct std::tuple_size<Tuple<Ts...>> :
    std::integral_constant<std::size_t, sizeof...(Ts)> {};

// Specialization for std::tuple_element
template<std::size_t I, typename... Ts>
struct std::tuple_element<I, Tuple<Ts...>> {
    static constexpr std::array types = { ^^Ts... };
    using type = [: types[I] :];
};

// Get the nth field in a type using reflection
consteval std::meta::info get_nth_field(
    std::meta::info r, std::size_t n) {
    return nonstatic_data_members_of(r)[n];
}

// Access the I-th element of a Tuple (non-const reference)
template<std::size_t I, typename... Ts>
constexpr auto get(Tuple<Ts...>& t) noexcept
```

```
    -> std::tuple_element_t<I, Tuple<Ts...>>& {
    return t.data.[:get_nth_field(^^decltype(t.data), I):];
}

// Access the I-th element of a Tuple (const reference)
template<std::size_t I, typename... Ts>
constexpr auto get(Tuple<Ts...> const& t) noexcept
    -> std::tuple_element_t<I, Tuple<Ts...>> const & {
    return t.data.[:
        get_nth_field(^^decltype(t.data), I):];
}

// Access the I-th element of a Tuple (rvalue reference)
template<std::size_t I, typename... Ts>
constexpr auto get(Tuple<Ts...>&& t) noexcept
    -> std::tuple_element_t<I, Tuple<Ts...>> && {
    return std::move(t.data).[:
        get_nth_field(^^decltype(t.data), I):];
}

int main() {
    auto [x, y, z] = Tuple{ 1, 'c', 3.14 };
    assert(x == 1);
    assert(y == 'c');
    assert(z == 3.14);
}
```

能这样实现的关键就在于代码生成，以前没有反射，只能通过递归继承或是递归复合来实现 `Tuple`，而后者又存在诸多问题，一般都是采用前者实现。通过 `define_aggregate`，可以直接合成一个 `storage` 内部类，所有 `Tuple` 元素全部通过 `data_member_spec` 注入到该内部类中，便可以轻易地生成一个 `Tuple` 类。

借助反射，`std::tuple_element` 特化也变得非常简单，只需将反射元信息序列存入一个静态数组，再利用 Splicers 取得指定索引的类型即可。索引式访问的 `get()` 实现亦是轻而易举，通过 `nonstatic_data_members_of()` 元函数可以直接得到某个元对象的所有非静态数据成员，再拼接访问指定索引的成员，就

实现了之前需要强制类型转换才能实现的功能[16]。

实现了 std::tuple_size、std::tuple_element 和自定义的 get 函数，自定义的 Tuple 类型便可以支持 Structure Bindings。这就是以上代码所涉及的核心知识点。

9.11.7 Struct to Struct of Arrays

这个例子采用代码生成将结构体的非静态数据成员转变成指定大小的数组形式，如：

```
struct X { int val; };
using Xs = struct_of_arrays<X, 10>;
// equivalent to:
// struct Xs {
//   std::array<int, 10> val;
// };
```

利用反射得到结构体的所有成员信息，再利用这些信息生成新的类型，这是操纵类型的经典运用。

struct_of_arrays 便需要使用代码生成相关的元函数才能实现，下面展示了具体的实现，同时也是一个例子。

```
template<typename T, std::size_t N>
struct struct_of_arrays_impl;

consteval auto make_struct_of_arrays(
    std::meta::info type, std::meta::info N)
    -> std::meta::info {
    auto old_members = nonstatic_data_members_of(type);
    std::vector<std::meta::info> new_members = {};
    for (std::meta::info member : old_members) {
        auto array_type = substitute(^^std::array, {
            type_of(member), N });
        auto mem_descr = data_member_spec(array_type, {
            .name = identifier_of(member) });
        new_members.push_back(mem_descr);
```

[16]具体实现见 https://www.cppmore.com/2020/04/25/understanding-variadic-templates/。

```
    }
    return define_aggregate(
        substitute(^^struct_of_arrays_impl, { type, N }),
        new_members);
}

template<typename T, std::size_t N>
using struct_of_arrays = [:make_struct_of_arrays(^^T, ^^N):];

struct point {
    float x;
    float y;
    float z;
};

int main() {
    using points = struct_of_arrays<point, 2>;

    points p = {
        .x = { 1.1, 2.2 },
        .y = { 3.3, 4.4 },
        .z = { 5.5, 6.6 }
    };

    static_assert(p.x.size() == 2);
    static_assert(p.y.size() == 2);
    static_assert(p.z.size() == 2);

    for (auto i = 0uz; i != 2; ++i) {
        std::println("p[{}] = ({}, {}, {})", i,
            p.x[i], p.y[i], p.z[i]);
    }
}

// Output:
```

```
// p[0] = (1.1, 3.3, 5.5)
// p[1] = (2.2, 4.4, 6.6)
```

由原结构体生成的新结构体为 `struct_of_arrays_impl`，其模板参数通过 `substitute` 元函数与原结构体保持一致，其成员是通过迭代原结构体非静态数据成员，替换成员类型为 `std::array`，再以 `data_member_spec` 将新类型成员的名称与原名称保持一致而得到。实际定义新类型则是通过 `define_aggregate` 实现。

如此，便实现了一个通用的结构体转数组结构体的功能。

9.11.8 Compile-Time Ticket Counter

在 §6.6，我们展示过利用 Friend Injection 技巧实现编译期计数器的例子，如何用静态反射实现类似的功能呢？下面是一个例子：

```
class TU_Ticket {
    template<int N> struct Helper;
public:
    static consteval int next() {
        int k = 0;

        // Search for the next incomplete 'Helper<k>'.
        std::meta::info r;
        while (!is_incomplete_type(r = substitute(
            ^^Helper, { std::meta::reflect_value(k) })))
            ++k;

        // Define 'Helper<k>' and return its index.
        define_aggregate(r, {});
        return k;
    }
};

int main() {
    int x = TU_Ticket::next();  // x initialized to 0.
    int y = TU_Ticket::next();  // y initialized to 1.
    int z = TU_Ticket::next();  // z initialized to 2.
```

```
23
24      std::cout << "x=" << x << '\n';
25      std::cout << "y=" << y << '\n';
26      std::cout << "z=" << z << '\n';
27  }
```

原理很简单，`Helper` 类模板只有声明，并没有定义，每次调用 `next()` 时，以 `substitute()` 和 `std::meta::reflect_value()` 这两个元函数来指定 `Helper` 的模板参数，构造模板特化；起初模板特化没有定义，所以直接跳出循环，接着采用 `define_aggregate()` 元函数来生成定义，于是下一次循环就不会直接跳出，而且不断增加 `k` 值，生成新的模板特化定义。

与 Friend-Injection 技巧相比，这种实现要直观许多，但更好的方式，当然还是直接由标准支持编译期变量来得简单。

9.11.9 Pack Indexing with Reflection

利用反射，也可以实现索引式访问参数包。代码如下：

```
1  template<size_t I, typename... Args>
2  consteval auto pack_index(const Args&... args) {
3      constexpr std::array params = { ^^args... };
4      return [: params[I] :];
5  }
6
7  int main() {
8      static_assert(pack_index<0>(3, "ch", 2.0) == 3);
9      static_assert(pack_index<1>(3, "ch", 2.0) == "ch");
10     static_assert(pack_index<2>(3, "ch", 2.0) == 2.0);
11 }
```

所有参数都可以借助 Reflection Operator 得到一个 `std::meta::info` 元对象，于是便可以将它们装到 `std::array` 静态容器之中，利用容器的索引式访问来实现参数包的索引式访问。

9.11.10 Automatically Generating SQL Statements from A Struct

本节展示如何利用静态反射自动从结构体生成 SQL 语句。

首先，创建产生数据表的函数，代码如下：

```
template<std::meta::info Class>
consteval auto create_table() {
    auto table_name = identifier_of(Class);
    auto sql = "CREATE TABLE " + std::string(table_name) +
               "(";
    bool first_seen = false;
    for (std::meta::info member :
        nonstatic_data_members_of(Class)) {
        if (first_seen) {
            sql += ", ";
        }

        sql += create_column(member);
        first_seen = true;
    }

    return sql + ");";
};
```

Class 就是结构体的反射对象，表名和结构体名称保持一致，通过 identifier_of() 获取，列名和成员变量保持一致，遍历后交由 create_column() 实际生成：

```
consteval auto create_column(std::meta::info member) {
    return std::string(identifier_of(member)) + " " +
        to_sql(type_of(member));
}
```

该函数同样利用 identifier_of() 元函数得到成员变量的名称，再通过 type_of() 元函数得到其类型元信息，交由 to_sql() 实际生成：

```
consteval const char* to_sql(std::meta::info type) {
    if(type == ^^int) return "INTEGER";
    else if(type == ^^std::string) return "TEXT";
    else return "UNKNOWN_TYPE";
}
```

成员变量的类型与 SQL 类型一一映射，返回映射的名称。

测试代码为：

```
struct person {
    int age;
    std::string name;
};

int main() {
    static_assert(create_table<^^person>() ==
        "CREATE TABLE person(age INTEGER, name TEXT);");
}
```

注意 `create_table()` 返回类型为 `std::string`，目前只支持 Transient Allocation，其结果只能在常量求值阶段使用。若想在运行期使用该结果，可以使用第八章中介绍的技巧。

9.11.11 Parsing Command-Line Options

再来看一个利用反射仿 Rust clap(Command Line Argument Parser) 的实现，clap 是 Rust 的命令行参数解析器。

```
struct Args : Clap {
    Option<std::string, {
        .use_short = true, .use_long = true }> name;
    Option<int, {
        .use_short = true, .use_long = true }> count = 1;
};

int main(int argc, char** argv) {
    auto opts = Args{}.parse(argc, argv);

    // opts.count has type int
    for (int i = 0; i < opts.count; ++i) {
        // opts.name has type std::string
        std::println("Hello {}!", opts.name);
    }
}
```

例子中定制的 Args 支持两种参数，一个是 name，一个是 count，其默认值为 1。如果编译参数为：

```
./test -n WG21 -c 7
```

-n 就对应于 name，而 -c 对应于 count。那么输出结果将为：

```
Hello WG21!
Hello WG21!
Hello WG21!
Hello WG21!
Hello WG21!
Hello WG21!
Hello WG21!
```

可以在 Args 中定制自己的参数列表，所有的解析操作都封装在 Clap 当中。要实现这样的效果，首先需要定义 Flags 和 Option：

```
struct Flags {
    bool use_short;
    bool use_long;
};

template<typename T, Flags flags>
struct Option {
    std::optional<T> initializer;

    Option() = default;
    Option(T t) : initializer(t) {}

    static constexpr bool use_short = flags.use_short;
    static constexpr bool use_long  = flags.use_long;
};
```

Flags 表示参数的形式，比如短形式为 -n，长形式就为 --name，根据不同的形式进行不同方式的解析；Option 表示定制的可选参数，具有两个构造函数，故而参数值的初始化是可选的。例如，只写 ./test -n WG21，此时 count 提供默认初始化为 1，从而简化参数。

其次，定义解析方式 Clap：

```
struct Clap {
    template<typename Spec>
    auto parse(this Spec const& spec,
        int argc, char** argv) {
        // ...
    }
};
```

这里使用 C++23 Deducing this 替换了传统的 CRTP 作为定制点表示方式，以简化代码。`argc` 和 `argv` 被传递进来，下一步操作：

```
template<typename Spec>
auto Clap::parse(this Spec const& spec,
    int argc, char** argv) {
    std::vector<std::string_view> cmdline(
        argv + 1, argv + argc);

    // check if cmdline contains --help, etc.

    struct Opts;
    static_assert(is_type(spec_to_opts(^^Opts, ^^Spec)));
    Opts opts;

    // ...
```

如果参数列表为 `./test -n WG21 -c 7`，那么除了第一个参数，剩余的实际参数都被保存到了 `cmdline` 之中，所以 `cmdline` 的大小就为 4。

紧接着开始解析，先通过代码生成自动产生 `Opts` 类，这个类作为解析的结果，也就是 `auto opts = Args{}.parse(argc, argv);` 中的 `opts` 类型。这个返回类型根据用户自定义的 `Args` 类中的非静态数据成员自动生成。对于当前的例子，生成后的结构为：

```
struct Opts { std::string name; int count; };
```

实际生成工作由 `spec_to_opts` 完成：

```
consteval auto spec_to_opts(
    std::meta::info opts, std::meta::info spec)
```

```
    -> std::meta::info {
    std::vector<std::meta::info> new_members;
    for (auto member : nonstatic_data_members_of(spec)) {
        auto new_type =
            template_arguments_of(type_of(member))[0];
        new_members.push_back(data_member_spec(new_type, {
            .name = identifier_of(member) }));
    }
    return define_aggregate(opts, new_members);
}
```

逻辑不算复杂，就是使用 `data_member_spec` 和 `define_aggregate` 代码生成元函数来完成简单类型的注入工作。

下一步，需要借助新类型 `Z` 和 `expand()` 来进行参数遍历：

```
template<typename Spec>
auto Clap::parse(this Spec const& spec,
    int argc, char** argv) {
    // ...

    struct Z {
        std::meta::info spec;
        std::meta::info opt;
    };

    [:expand([]() consteval {
        auto spec_members =
            nonstatic_data_members_of(^^Spec);
        auto opts_members =
            nonstatic_data_members_of(^^Opts);

        std::vector<Z> v;
        for (size_t i = 0; i != spec_members.size(); ++i) {
            v.push_back({
                .spec = spec_members[i],
                .opt  = opts_members[i] });
```

```
        }
        return v;
    }()):] >> [&]<auto Z> {
        // ...
    }
```

`Z` 包含两个成员，分别保存 `Args` 和 `Opts` 的非静态数据成员信息，当前示例，其大小为 2。每一组信息就对应一个参数，2 个分别对应 `-n` 和 `-c`。

如果用 Expansion Statements 来写，逻辑会更加清晰，对应的写法为：

```
template for (constexpr auto [sm, om] :
    std::views::zip(nonstatic_data_members_of(^^Spec),
                    nonstatic_data_members_of(^^Opts))) {
    // ...
}
```

具体处理每一组参数的逻辑如下：

```
template<typename Spec>
auto Clap::parse(this Spec const& spec,
    int argc, char** argv) {
    // ...
    >> [&]<auto Z> {
        constexpr auto sm = Z.spec;
        constexpr auto om = Z.opt;

        auto& cur = spec.[:sm:];

        // find the argument associated with this option
        auto it = std::find_if(
            cmdline.begin(), cmdline.end(),
            [&](std::string_view arg) {
                return cur.use_short &&
                    arg.size() == 2 &&
                    arg[0] == '-' &&
                    arg[1] == identifier_of(sm)[0] ||
                    cur.use_long &&
```

```
                    arg.starts_with("--") &&
                    arg.substr(2) == identifier_of(sm);
        });

        if (it == cmdline.end()) {
            // no such argument
            if constexpr (
                has_template_arguments(type_of(om)) &&
                template_of(type_of(om)) == ^^std::optional){
                return;
            } else if (cur.initializer) {
                opts.[:om:] = *cur.initializer;
                return;
            } else {
                std::exit(EXIT_FAILURE);
            }
        } else if (it + 1 == cmdline.end()) {
            std::exit(EXIT_FAILURE);
        }

        // alright, found our argument, try to parse it
        auto iss = std::ispanstream(it[1]);
        if (iss >> opts.[:om:]; !iss) {
            std::exit(EXIT_FAILURE);
        }
    };

    return opts;
}
```

整体实现思路就是根据 `cur` 中的信息在参数列表 `cmdline` 中查找，如果没有查到，则看参数是否可选，有可选默认值的，把该值读取出来，保存到 `opts` 中；如果查找到的位置后面没有紧跟参数值，如 `-n` 后面什么也没有，则表示缺少参数值。

如果找到了参数，则使用 C++23 `std::ispanstream` 将值读取到 `opts` 返回值当中，`it` 查找到的位置为参数的位置，参数位置后面 `it[1]` 就是参数值的位置。

如此，便借助反射实现了一个可定制的 `Clap`，逻辑还是比较清晰的，但受限于当前的实现，绕了一些路，显得麻烦了一些。

完整实现为：

```
// EDG: https://godbolt.org/z/nxY59TfGM
struct Flags {
    bool use_short;
    bool use_long;
};

template<typename T, Flags flags>
struct Option {
    std::optional<T> initializer;

    Option() = default;
    Option(T t) : initializer(t) { }

    static constexpr bool use_short = flags.use_short;
    static constexpr bool use_long  = flags.use_long;
};

consteval auto spec_to_opts(
    std::meta::info opts, std::meta::info spec)
    -> std::meta::info {
    std::vector<std::meta::info> new_members;
    for (auto member : nonstatic_data_members_of(spec)) {
        auto new_type =
            template_arguments_of(type_of(member))[0];
        new_members.push_back(data_member_spec(new_type, {
            .name = identifier_of(member) }));
    }
    return define_aggregate(opts, new_members);
}

struct Clap {
    template<typename Spec>
```

```
33      auto parse(this Spec const& spec,
34          int argc, char** argv) {
35          std::vector<std::string_view> cmdline(
36              argv + 1, argv + argc);
37
38          // check if cmdline contains --help, etc.
39
40          struct Opts;
41          static_assert(is_type(spec_to_opts(^^Opts, ^^Spec)));
42          Opts opts;
43
44          struct Z {
45              std::meta::info spec;
46              std::meta::info opt;
47          };
48
49          [:expand([]() consteval {
50              auto spec_members =
51                  nonstatic_data_members_of(^^Spec);
52              auto opts_members =
53                  nonstatic_data_members_of(^^Opts);
54
55              std::vector<Z> v;
56              for (size_t i = 0; i != spec_members.size();
57                  ++i) {
58                  v.push_back({
59                      .spec = spec_members[i],
60                      .opt  = opts_members[i] });
61              }
62              return v;
63          }()):] >> [&]<auto Z> {
64              constexpr auto sm = Z.spec;
65              constexpr auto om = Z.opt;
66
67              auto& cur = spec.[:sm:];
```

```
68
69              // find the argument associated with this option
70              auto it = std::find_if(
71                  cmdline.begin(), cmdline.end(),
72                  [&](std::string_view arg) {
73                      return cur.use_short &&
74                          arg.size() == 2 &&
75                          arg[0] == '-' &&
76                          arg[1] == identifier_of(sm)[0] ||
77                          cur.use_long &&
78                          arg.starts_with("--") &&
79                          arg.substr(2) == identifier_of(sm);
80                  });
81
82              if (it == cmdline.end()) {
83                  // no such argument
84                  if constexpr (
85                      has_template_arguments(type_of(om)) &&
86                      template_of(type_of(om)) ==
87                      ^^std::optional) {
88                      return;
89                  } else if (cur.initializer) {
90                      opts.[:om:] = *cur.initializer;
91                      return;
92                  } else {
93                      std::exit(EXIT_FAILURE);
94                  }
95              } else if (it + 1 == cmdline.end()) {
96                  std::exit(EXIT_FAILURE);
97              }
98
99              // alright, found our argument, try to parse it
100             auto iss = std::ispanstream(it[1]);
101             if (iss >> opts.[:om:]; !iss) {
102                 std::exit(EXIT_FAILURE);
```

```
            }
        };

        return opts;
    }
};

struct Args : Clap {
    Option<std::string, {
        .use_short = true, .use_long = true }> name;
    Option<int, {
        .use_short = true, .use_long = true }> count = 1;
};

int main(int argc, char** argv) {
    auto opts = Args{}.parse(argc, argv);

    for (int i = 0; i < opts.count; ++i) {
        std::println("Hello {}!", opts.name);
    }
}
```

9.12 Limitations

纵使静态反射顺利进入 C++26，也不会是完全体，存在一些限制。

比如，对于 Splicers，无法在任何上下文中拼接构造函数的反射元对象，也无法在模板声明中拼接 Concepts 的反射元对象，将类成员元对象拼接为 Designated Initializers 中的 Designators 亦不支持……

Expansion Statements 和 Source Code Injection 暂时的替代方式也略显笨重，代码注入能力弱得可怜。并且，目前静态反射无法观察某个翻译单元内的所有类型信息，也无法获取 Attributes 元信息，这也使某些极有用处的场景难以实现。

一波动而万波随，初始版本稳定后，相信这些限制会在快速迭代中不断消除，逐渐完善整个反射元编程系统。

9.13 Summary

本章系统介绍了一下 C++26 准备加入的静态反射（如果一切顺利），这也是下一版本中的大件之一，将使 C++ 元编程步入下一个崭新的阶段。

从静态反射的历史、概念、语法，到元编程库、代码注入，再到应用示例，带大家提前纵览了一下 C++ 的反射元编程能力。反射能够读取和操纵类型元信息，动态调整程序自身的结构和行为，这是其他元编程工具所不具备的功能，却正是我们一直想要的功能。

代码注入是产生式元编程最核心的武器，宏、模板、折叠表达式、常量表达式等等方式，虽然也能够生成代码，但可控性太低。代码注入配合静态反射，能够直接改变程序结构，产生想要的代码，真正达到编程源码本身，写出抽象层次更高、稳定性更好、适应性更强的代码。只是，本章中的代码注入依然无法实现这种目标，在下一章，我们将来看几个专为源码注入而设计的特性，它们能够丰富和加强反射元编程的代码生成能力。

Chapter 10

Source Code Injection

什么是源码注入？为什么它如此重要呢？

虽然反射很重要，但我们真正需要的是反射元编程——能够支持产生式元编程——否则也只若是良弓缺弦。尽管静态反射提供了 `define_aggregate()` 和 `data_member_spec()` 这两个具有基本代码注入能力的元函数，但和真正的源码注入相比，当前所能提供的产生式元编程能力还要差上十万八千里呢。

因此，最后一章，便来纵览一下未来产生式元编程最强大的特性——源码注入，这也是 C++ 第三阶段元编程的核心特性，能够替代许多前两个阶段元编程的老特性，简化元编程流程，编写更安全的代码。

当前正处混沌时期，本章将介绍许多与代码注入相关的特性，它们很难进入 C++26，也可能只是昙花一现，但未尝不是发展道路上的宝贵积淀，某些部分肯定会在未来成为 C++ 反射元编程的重要成分。

10.1 Overview

目前，静态反射基于 `define_aggregate()` 元函数，搭配一系列 Spec 成员规范元函数来满足简单的代码注入需求。`data_member_spec()` 就是一个 Spec API，负责注入数据成员，未来可能还会增加针对命名空间、枚举、函数、任意块代码的其他版本，以满足更多注入需求。只是，`define_aggregate()` 的调用是一个表达式，并且注入过程是一个整体，无法分步注入单个部分，所以，有时得借助 `static_assert` 来辅助完成注入功能，例如：

```
1 struct point;
2 static_assert(is_type(define_aggregate(^^point, {
```

```
    data_member_spec(^^int, { .name = "x" }),
    data_member_spec(^^int, { .name = "y" })
}));

int main() {
    point p{ 1, 2 };
    // x: 1, y: 2
    std::println("x: {}, y: {}", p.x, p.y);
}
```

这种注入方式称为 Spec API，Spec 是 Specification（规范）的缩写，表示某种用于描述或定义类、成员变量、函数等特性的规范对象。

这只是比较弱的注入模型，还有两种注入模型值得关注，一种是 P1717R0 最初引入的 Fragments，另一种是 P3294R2 提出的更新一点儿的 Token Sequences。

同样的例子，用 Fragments 模型注入的写法为：

```
struct point {
    consteval {
        -> fragment struct {
            int x;
            int y;
        };
    }
};
```

而 Token sequences 模型对应的写法为：

```
struct point {
    consteval {
        queue_injection(^{ int \id("x"sv); });
        queue_injection(^{ int \id("y"sv); });
    }
};
```

即便当前大家还不知道这两种代码注入模型的具体概念，却也能够对比一下它们的注入方式，建立一个初始印象。

接着，便来详细介绍一下不同注入模型的基本知识点。

10.2 Fragments

先来看 Fragments，还是以一个简单的例子开始：

```
struct X {
    consteval {
        for (int num = 0; num < 10; ++num)
            -> fragment struct {
                int unqualid("data_", %{num});
            };
    }
};
```

此处便使用源码注入，自动生成了以下代码：

```
struct X {
    int data_0;
    int data_1;
    int data_2;
    int data_3;
    int data_4;
    int data_5;
    int data_6;
    int data_7;
    int data_8;
    int data_9;
};
```

要进行注入的代码区域，称为 *metaprogram*，语法如下：

```
consteval {
    ...
};
```

可以将其当作是一个不需要参数的 `consteval` 函数，编译时会自动执行。

metaprogram 中的一些操作并不需要注入，例子中利用 `for` 自动产生 `num` 的代码便是此列，真正需要注入的语句得通过注入语句的语法 “->” 指定，指定的代码才会注入到源码中去。

紧跟着注入语句的代码片段称为 *fragments*，是一个表达式，类型为元对象（`std::meta::info`）。代码片段有许多种类，如 *namespace fragments*, *class fragments*, *enumeration fragments*, *block fragments*，顾名思义，分别表示注入到相应上下文的源码中去。例子中使用的 `fragment struct`，就属于 *class fragments*，也存在 `fragment class`，区别与 `struct` 和 `class` 的区别一样。

需要注意，class fragments 并不用于描述一个真正的类，它仅仅是描述类中的成员。也就是说，不用给这个类起名字，起了也会被替换掉，最终被注入的只是这个类的 *body*。并且，class fragments 只能注入到类里面。

那么如何在 fragments 中引入外部的变量呢？

这就需要使用 *unquote operator* 了，语法为“`%{ ... }`”，它允许引用局部变量作为 fragments 中的变量名或类型名。而 *unqualid operator*，可以让我们从字符串组合新的代码。

前文提到，fragments 的类型为元对象，那么其实是可以分开定义的，看下面这个 enumeration fragments 的例子：

```
constexpr meta::info rgb = fragment enum {
    red, blue, green
};

enum class color {
    consteval -> rgb
};
```

这里会把 fragments 中的值注入到枚举类型的源码中去，metaprogram 只有一行的情况下，可以省略“`{}`”。因此，源码注入后产生的代码如下：

```
enum class color {
    red,
    blue,
    green
};
```

因为反射的类型也是元对象，所以可以配合起来使用，实现更多强大的源码注入能力。看个简单的例子：

```
struct A {
    void foo() {}
};
```

```

struct X {
    consteval -> ^^A::foo;
};
```

这样就轻松地将 A 的成员函数，注入到了 X 当中。

同时，也可以在注入的过程中，修改原始定义，比如：

```
struct X {
    consteval {
        std::meta::info foo_refl = ^^A::foo;
        make_constexpr(foo_refl);

        const char* name = identifier_of(foo_refl);
        set_new_name(foo_refl,
            __concatenate("constexpr_", name));

        -> foo_refl;
    }
};
```

这便相当于以下声明：

```
struct X {
    constexpr void constexpr_foo() {}
};
```

可以看到，反射配合源码注入，具备强大的产生式元编程能力。

以上便是 Fragments 最核心的注入功能，在 Use Cases 一节，会将其与 Token Sequences 这种更新的注入模型作对比，展示更多复杂用法。

10.3 Token Sequences

P3294R2 认为：代码生成的本质在于将一些孤立或几乎无意义的片段组装成有意义的结构，对生成代码进行早期语法和语义验证反而是一种整体负担，通过拼接的方式生成代码，并将语法/语义分析推迟到注入时进行，才是代码生成最简单、最直接的方法。

因此，他们决定以 Token Sequences（标记序列）作为生成代码的核心构建模块，未解析的 Token Sequences 允许灵活的组合，同时将语义分析推迟到注入时再进行，以避免在组合时重建注入点上下文的复杂性。

Fragments 中的例子，用 Token Sequences 实现如下：

```
struct X {
    consteval {
        for (int num = 0; num < 10; ++num) {
            queue_injection(^{ int \id("data_"sv, num); });
        }
    }
};
```

其中，“`^{ ... }`”称为 Token Sequenece 表达式，类型为 `std::meta::info`，里面的代码就是可以注入的代码，而实际注入则通过一个新的 `queue_injection()` 元函数来实现。为了访问外部变量，P3294R2 引入了三个插值器：

1. `\(expression)`：值插值器，插入的是表达式的值，配合 Splicers `[:\(r):]` 使用可以插入类型，此插值器中，字符串将还是会被注入为字符串，而不会像标识符插值器那样，注入为标识符；
2. `\id(string, string-or-intopt...)`：标识符插值器，参数会自动拼接，并将结果作为标识符名称；
3. `\tokens(expression)`：标记序列插值器，拼接其他 Token Sequences 的内容。

例子中要注入的是变量名称，因此使用标识符插值器，它会自动拼接所有参数。

如果要拼接多个 Token Sequences，需要使用标记序列插值器，如：

```
constexpr auto t1 = ^{ abc };
constexpr auto t2 = ^{ def };
constexpr auto t3 = ^{ \tokens(t1) \tokens(t2) };
// exposition only
static_assert(t3 != ^{ abcdef });
static_assert(t3 == ^{ abc def });

constexpr auto t4 = ^{ hello  = /* world */   "world" };
```

```
constexpr auto t5 = ^{ /* again */ hello="world" };
// exposition only
static_assert(t4 == t5);
```

整体而言，插值器的使用不算复杂，多余的空格和注释在解析时也会被丢弃，没有什么奇异的行为，不多赘述。

除了 `queue_injection(e)`，还存在一个注入元函数 `namespace_inject(ns, e)`，前者会依序将标记序列 `e` 注入到当前常量求值（即 `consteval` 块）的末尾，而后者会立即将标记序列注入到 `ns` 指定的命名空间中。下面是一个例子：

```
consteval auto f(std::meta::info r, int val,
    std::string_view name) {
    return ^{ constexpr [:\(r):] \id(name) = \(val); };
}

constexpr auto r = f(^^int, 42, "x");

namespace N {}

consteval {
    // this static assertion will be injected
    // at the end of the block
    queue_injection(^{ static_assert(N::x == 42); });

    // this declaration will be injected right
    // into ::N right now
    namespace_inject(^N, r);
}

int main() {
    return N::x != 42;
}
```

这是最重要的是要清楚地意识到：`namespace_inject()` 是立即注入的，而 `queue_injection()` 是排队注入到当前 `consteval` 块的末尾。否则会对代码注入的前后逻辑产生疑惑。

Token Sequences 的核心内容就是以上这些，更多用法同样在 Use Cases 一节展示。

10.4 Scoped Macros

宏允许简单的代码替换，甚至不需要对输入进行评估和解析，只需原封不动地替换成目标代码。在前四章我们已深入讲解过宏元编程的高级知识，由于宏并非是图灵完整的，设计上存在很大缺陷，即使是像条件分支、循环、递归这些简单的操作，都需要具备极其深入的理解才能够实现，就算实现了也有嵌套深度限制，而且宏也没有作用域限制，可谓是难用至极。

P3294R2 同时带来了一种替代宏的方式，可以把宏作为一个接收并返回 Token Sequences 的函数，再在调用点通过某种语法标记自动注入结果。

一个简单的例子：

```
consteval auto fwd(meta::info x) -> meta::info {
    return ^{
        static_cast<
            decltype([:\tokens(x):])&&>([:\tokens(x):]);
    };
}

using std::forward;
auto old_f = [](auto&& x) { return forward<decltype(x)>(x);};
auto new_f = [](auto&& x) { return fwd!(x); };
```

该例子利用新的宏机制来自动生成 `std::forward` 转发代码，本身意义不大，主要是为了展示其用法。

其中，`fwd!(x)` 是 `immediately_inject(fwd(^{ x }))` 的语法糖，这是一种将代码注入到表达式中的新机制。！这种语法是从 Rust 借鉴而来，表示这是一个宏的注入操作。`fwd` 和常规的函数相同，变量访问也需要满足作用域规则，可以避免传统宏中存在的相关问题，故名 Scoped Macros。

再来看提案中的一个例子：

```
consteval auto assert_eq(meta::info a, meta::info b)
    -> meta::info {
    return ^{
        do {
```

```
            auto sa = \(stringify(a));
            auto va = \tokens(a);

            auto sb = \(stringify(b));
            auto vb = \tokens(b);

            if (not (va == vb)) {
                std::println(
                    stderr,
                    "{} ({}) == {} ({}) failed at {}",
                    sa, va,
                    sb, vb,
                    \(source_location_of(a)));
                std::abort();
            }
        } while (false);
    };
}
```

`stringify` 用于将参数字符串化，`source_location_of` 返回错误的源码位置，都只是处于构想中的元函数。断言的实现逻辑很简单，结果不相等时利用反射获取的信息，报告错误且终止程序。`do { ... } while (false)` 则是宏定义的惯用法，可以确保宏的行为像一个单一的语句，避免潜在的语法问题。

如果调用 `assert_eq!(42, factorial(3))`，将自动生成以下代码：

```
do {
    auto sa = "42";
    auto va = 42;

    auto sb = "factorial(3)";
    auto vb = factorial(3);

    if (not (va == vb)) {
        std::println(
            stderr,
            "{} ({}) == {} ({}) failed at {}",
```

```
            sa, va,
            sb, vb,
            /* some source location */);
        std::abort();
    }
} while(false);
```

这功能其实用 C Macros 也能够实现，但 Scoped Macros 要更加易读且易用一些。

需要注意，C Macros 其实是一种 Unhygienic Macros（非卫生宏），无法避免宏展开过程中引入意外的名称冲突或作用域问题。举个例子[1]：

```
#define INCI(i) do { int x = 0; ++i; } while(0)

int main() {
    int x = 4, y = 8;

    // macro called first time
    INCI(x);

    // macro called second time
    INCI(y);

    // x = 4, y = 9
    std::cout << "x = " << x << ", y = " << y;
}
```

以上代码中的 x 存在名称冲突，main 中的 x 和宏内部的 x 名称相同，所以结果不如预期。

如果 assert_eq 也是此类宏，那么调用 assert_eq!(42, sa * 2) 也会出现名称冲突。而 Scoped Macros 可以通过一些方式设计为 Hygienic Macros（卫生宏），比如使宏内的名称具有某种唯一性：

```
consteval auto assert_eq(meta::info a, meta::info b)
    -> meta::info {
    auto [sa, va, sb, vb] =
```

[1] borrowed from https://www.geeksforgeeks.org/hygienic-macros-introduction/

```
        std::meta::make_unique_names<4>();

    return ^{
        do {
            auto \id(sa) = \(stringify(a));
            auto \id(va) = \tokens(a);

            auto \id(sb) = \(stringify(b));
            auto \id(vb) = \tokens(b);

            if (not (\id(va) == \id(vb))) {
                std::println(
                    stderr,
                    "{} ({}) == {} ({}) failed at {}",
                    \id(sa), \id(va),
                    \id(sb), \id(vb),
                    \(source_location_of(a)));
                std::abort();
            }
        } while (false);
    };
}
```

引入一个 `make_unique_names()` 元函数，用来产生指定数量的唯一名称，不过使用时需要通过 `\id()` 来构建标识符名称。或者，也可以在内部采取某种机制，令 Scoped Macros 默认就为 Hygienic，此时便不需要单独整一个元函数，再每次通过插入符来定义唯一名称了。

但是，并不是说 Unhygienic Macros 就没有必要，少数情境下反而需要名称相同，例如下面的指令宏：

```
consteval auto λ(meta::info body) -> meta::info {
    return ^{
        [&](auto&& it)
            noexcept(noexcept(\tokens(body)))
            -> decltype(\tokens(body))
        {
```

```
            return \tokens(body);
        }
    }
}

auto positive = std::ranges::count_if(r, λ!(it > 0));
```

这里需要利用参数中传递的名称来简化 Lambda 调用，`it` 名称需要和宏内的 `it` 名称一致。因此，最后 Scoped Macros 是采取默认 Hygienic，还是默认 Unhygienic，再结合一个元函数来生成唯一的名称，抑或是其他方式，且看后续的发展。

目前该特性只是处于初始阶段，也没有编译器支持，未来变化可能很大，所以不多讨论。

10.5 Annotations

反射注解（即 C++ 中的 Attributes）是一个相当强大的特性，如果使用过类似 Java Spring 这种框架，肯定对此深有体会。关键是先要支持自定义 Attributes，再允许反射自定义的这些属性值。

P1887R1([2]) 提供了一个大致的概览，其中的例子如下：

```
namespace Catch {
    struct [[decorator]] test_case {
        constexpr test_case(const char* name)
            : name(name) { }
        const char* name;
    };
}

namespace test {
    void g() {}

    [[+Catch::test_case("Printing something")]]
    void f() {
        g();
```

[2] https://wg21.link/p1887r1

```
        std::cout << "Hello World\n";
    }
}

int main () {
    static constexpr auto r = meta::range(reflexpr(test));
    template for (constexpr auto member : r) {
        static constexpr bool b =
            meta::has_attribute<Catch::test_case>(member);
        if constexpr(meta::is_function(member) && b) {
            constexpr auto data =
                meta::attribute<Catch::test_case>(member);
            std::cout << "Running test " << data.name <<
↪   "\n";
        }
    }
}
```

以 `[[decorator]]` 表示自定义 Attributes 的类型，这个类型可以通过 `+` 在声明 Attributes 时使用。再配合一些针对 Attributes 的元函数，能够遍历这些定义的属性值，如此一来，便可以通过该值来过滤一些函数，或是形成某种逻辑映射。

反射注解对于库设计来说很有用，但目前没有一点浪花，在此也不便多讲，大家随便看一下就行。

10.6 Use Cases

10.6.1 Automatically Generating Getters and Setters

本例子展示如何利用不同的注入模型配合反射为类型自动生成 Getters 和 Setters 成员函数。

当前有如下类：

```
struct book {
    std::string title;
    std::string author;
```

```
    int page_count;
};
```

想要的效果是自动生成如下代码：

```
struct book {
    std::string title;
    std::string author;
    int page_count;

    std::string get_title() const {
        return title;
    }

    void set_title(const std::string& title) {
        this->title = title;
    }

    std::string get_author() const {
        return author;
    }

    void set_author(const std::string& author) {
        this->author = author;
    }

    int get_page_count() const {
        return page_count;
    }

    void set_page_count(const int& page_count) {
        this->page_count = page_count;
    }
};
```

下面便来看不同注入模型如何完成这项任务。

10.6.1.1 Using Fragments

对于 Fragment 注入模型，首先，定义 gen_members() 函数：

```
consteval void gen_members(std::meta::info cls) {
    auto members =  nonstatic_data_members_of(cls);
    for (std::meta::info member : members) {
        gen_member(member);
    }
}
```

通过 nonstatic_data_members_of() 元函数遍历出类的所有非静态数据成员，再为每个数据成员生成 getter 和 setter 函数。gen_member() 函数的实现如下：

```
consteval void gen_member(std::meta::info m) {
    -> fragment struct {
        typename(type_of(%{m}))
        unqualid("get_", identifier_of(%{m}))() const {
            return unqualid(identifier_of(%{m}));
        }
    };

    -> fragment struct {
        void unqualid("set_", identifier_of(%{m}))(
        const typename(type_of(%{m}))&
        unqualid(identifier_of(%{m}))) {
            this->unqualid(identifier_of(%{m})) =
                unqualid(identifier_of(%{m}));
        }
    };
}
```

通过定义两个 fragments，利用元函数获取成员变量的名称、类型等信息，再借助 Lock3 提供的 unqualid() 拼接字符串作为标识符名称，就能够达到自动产生代码的能力。实际注入区域，通过 metaprogram 指定：

```
struct book {
    std::string title;
```

```
    std::string author;
    int page_count;

    consteval {
        gen_members(^^book);
    }
};
```

调用 `gen_members()` 为当前类中的非静态数据成员自动生成 Getters 和 Setters 函数，这个 fragments 具备可复用性，对所有的类都适用。

这种对当前类生成一些新代码，还有一种更推荐的方式，称为 *Metaclasses*，这个概念属于源码注入。上述例子以这种方式编写，代码如下：

```
consteval void gen_members(meta::info cls) {
    auto members =  nonstatic_data_members_of(cls);
    for (auto member : members) {
        ->member;

        gen_member(member);
    }
}

struct(gen_members) book {
    std::string title;
    std::string author;
    int page_count;
};
```

Metaclasses 就是一个 metaprogram，不过它强调的是从一个类的原型（*prototype*）来为该类产生新的代码，语法为 "`struct(...)`"，这样就无需在类内编写 metaprogram，这种方式要更加安全，编写的代码量也更少。

这里有一点需要注意，编译器在实例化该定义时，会有一个隐藏的 `self` 元对象，还有一个处于匿名空间之中的 `book` 类，这个就是原型类。所有声明的成员处于原型类中，`self` 当中并没有，因此得使用 `->member` 将这些成员重新产生出来，否则`book` 类当中找不到声明的那些成员。

10.6.1.2 Using Token Sequences

对于 Token Sequences 注入模型，代码生成的逻辑基本一致，但存在一点差异。主要表现在成员的注入形式上：

```
struct book {
    std::string title;
    std::string author;
    int page_count;

    // error: expression must have a constant value
    // note: invalid reflection for intrinsic metafunction
    consteval {
        std::meta::info cls = ^^book;
        auto members =
            std::meta::nonstatic_data_members_of(cls);
    }
};
```

以上方式不再可行，也许不是 Token Sequences 本身的问题，而是 EDG 实现的问题。总之，不能直接传递当前类本身到 metaprogram 中的元函数，实现必须变为：

```
struct book {
    consteval {
        gen_member(^^std::string, "title");
        gen_member(^^std::string, "author");
        gen_member(^^int, "page_count");
    }
};
```

不只 Getters 和 Setters，成员变量也得通过代码生成。此时，不再需要遍历当前类中的所有非静态数据成员，所以只需要 `gen_member()` 这一个函数：

```
consteval void gen_member(std::meta::info type,
    std::string_view name) {
    auto member = ^{ \id(name) };

```

```
    // Inject member
    queue_injection(^{ [:\(type):] \tokens(member); });

    // Inject getter for member
    queue_injection(^{
        [:\(type):] \id("get_"sv, name)() const {
            return \tokens(member);
        }
    });

    // Inject setter for member
    queue_injection(^{
        void \id("set_"sv, name)(
            const [:\(type):]& \tokens(member)) {
            this->\tokens(member) = \tokens(member);
        }
    });
}
```

注入操作，主要利用 Token Sequences 中的三种插值器，整体逻辑与 Fragments 版本一致。

10.6.2 Implementing Tuple<Ts...>

本书前面各章，展示过使用递归继承、Pack Indexing、Spec Injection API 等等实现 Tuple 的方法，本节再展示利用 Fragments 和 Token Sequences 注入模型的实现法。

10.6.2.1 Using Fragments

对于 Fragment 注入模型，实现起来不算复杂，代码为：

```
template<class... Ts>
struct my_tuple {
    consteval {
        // Lock3 does not support pack expansion here
        meta::info types[] = { reflexpr(Ts)... };
```

```
        for (size_t i = 0; i != sizeof...(Ts); ++i) {
            -> fragment struct {
                [[no_unique_address]] typename(meta::type_of(
                    %{types[i]})) unqualid("_", %{i});
            };
        }
    }
};

// CTAD
template<class... Ts>
my_tuple(Ts...) -> my_tuple<Ts...>;

int main() {
    my_tuple t{ 1, "hello", 2.1 };
    std::cout << t._0 << "\n"; // Output: 1
    std::cout << t._1 << "\n"; // Output: hello
    std::cout << t._2 << "\n"; // Output: 2.1
}
```

代码注入可以直接在类中生成多个成员变量，于是不必再利用各种技巧，简单而直接。

这里因为 Fragments 的实现较早，`reflexpr` 不支持参数包展开，但这只是实现的问题，不影响实现流程的表达。

10.6.2.2 Using Token Sequences

对于 Token Sequences 注入模型，实现起来更加简洁、清晰，对应的代码为：

```
template<class... Ts>
struct my_tuple {
    consteval {
        std::meta::info types[] = { ^^Ts... };
        for (auto i = 0uz; i != sizeof...(Ts); ++i) {
            queue_injection(^{ [[no_unique_address]]
                [:\(types[i]):] \id("_"sv, i); });
        }
```

```
    }
};

// CTAD
template<class... Ts>
my_tuple(Ts...) -> my_tuple<Ts...>;

int main() {
    my_tuple t{ 1, "hello", 2.1 };
    std::cout << t._0 << "\n"; // Output: 1
    std::cout << t._1 << "\n"; // Output: hello
    std::cout << t._2 << "\n"; // Output: 2.1
}
```

因为这个注入模型是今年提出的，和静态反射的最新特性紧密配合，有专门设计的插值语法，所以比 Fragments 简洁并不出奇，这种方式也不会存在参数包展开问题。

10.6.3 Implementing enable_if<B, T>

本节再来看一个生成类型别名的例子，以源码注入来实现 `enable_if`。

10.6.3.1 Using Fragments

`enable_if` 的核心就是根据条件确定是否定义类型，所以实现极为简单：

```
template<bool B, class T = void>
struct enable_if {
    consteval {
        if (B) {
            -> fragment struct {
                using type = T;
            };
        }
    }
};
```

只有当 `B` 条件为真时，才会向类中注入 `using type = T`。因此，这种代码生成能力，可以替代原本的模板偏特化，更加清楚地表达条件逻辑。

10.6.3.2 Using Token Sequences

Token Sequences 的实现逻辑相同，只是换了一种语法而已：

```
template<bool B, class T = void>
struct enable_if {
    consteval {
        if (B) {
            queue_injection(^{ using type = T; });
        }
    }
};
```

注入模型不同，但它们要支持的特性都是一样的，所以知道一个，就很容易转换为其他模型的写法。

10.7 Looking Ahead

不论哪种注入模型，要支持的功能其实都大差不差，包含其他相关特性，都是与反射并驾齐驱的，反射没正式进入标准，其他的肯定还要更晚。

C++26 还有数月，静态反射和 Spec 注入 API 的进展很突出，标准也很看重这个大件，进入的问题不大。至于其他注入模型、Scoped Macros、静态反射当前的局限、反射注解等等其他特性，最快也得等 C++29 了。

其他编译期特性对静态反射来说也非常重要，比如编译期函数参数，没有这个特性，用静态反射和源码注入来实现工厂模式都难以做到：

```
struct S {
    void print() {
        std::cout << "S::print()\n";
    }
};

struct factory {
    constexpr auto create(std::string_view name) const {
```

```
        consteval {
            // error: cannot use name in this context
            queue_injection(^{ return new \id(name); });
        }
    }
};

int main() {
    factory fty;
    auto s = fty.create("S");
    s->print();
}
```

如果能够支持这个功能，就能够彻底摆脱手动映射，实现更多强大的组件。总之，静态反射只是开始，未来还有许多工作。

10.8 Summary

本章介绍了反射元编程最核心的特性——源码注入，主要涉及 Fragments 和 Token Sequences 这两种注入模型，它们的目标是一致的，都是提供灵活的代码注入能力。至于 Scoped Macros 和 Annotations，它们也是反射元编程中的核心特性，只是优先级要比源码注入略低一些，等待时间也会更久一点。

反射元编程是产生式元编程最强大的一种工具，能够直接操纵类型、生成源码，具备可控性极强的代码生成能力。写库/框架时，这种特性可以从更底层去抽象逻辑，封装更多变化点，灵活性更强。

行文至此，本书也要结束了，提笔时是清秋遥夜，落笔时是冬景严凝，倏忽之间，已是一载。繁文写来费力，读来又岂是轻松，看到这里，不说满载而归，想来各位也是学到了不少元编程的高级技术，对产生式元编程建立了系统的认知。

落笔仓促，错漏难免，明年二月 C++26 特性就会最终确定，如有第二版，会一并修复补遗，那时想必许多人已经读到此处了吧 ~