Understanding Modern C++

发布 *1*

Darwin Yuan

2021 年 08 月 14 日

目录

1 引用 **1**

1.1 值与对象 . . . 1
1.2 别名 . . . 2
1.3 空间 . . . 2
1.4 受限的指针 . . . 3
1.5 左值 . . . 4

2 右值引用 **5**

2.1 缺失的拼图 . . . 5
2.2 **move** 语意 . . . 6
2.3 右值引用变量 . . . 7
2.4 速亡值 . . . 10

3 值与对象 **15**

3.1 值 . . . 15
3.2 对象 . . . 15
3.3 值与对象的关系 . . . 16
3.4 纯右值 . . . 16
3.5 泛左值 . . . 16

4 decltype **23**

4.1 有括号语意 . . . 24
4.2 无括号语意 . . . 25

5 auto 类型推演 **27**

5.1 **auto** 的语意 . . . 27
5.2 引用及 **const** . . . 28
5.3 指针 . . . 28

5.4 通用引用 . . . 29
5.5 初始化列表 . . . 30
5.6 **decltype(auto)** . . . 30
5.7 函数返回值类型的自动推演 . . . 31
5.8 非类型模版参数 . . . 31
5.9 函数模版的便捷写法 . . . 31

6 初始化 **33**
6.1 直接初始化 . . . 34
6.2 列表初始化 . . . 36
6.3 值初始化 . . . 37
6.4 默认初始化 . . . 38
6.5 拷贝初始化 . . . 38
6.6 零初始化 . . . 39

7 六大金刚 **41**
7.1 存在性 . . . 42
7.2 可操作性 . . . 46
7.3 平凡性 . . . 50

8 SFINAE **53**
8.1 函数重载 . . . 53

9 C++ 编译时元编程（一） **57**
9.1 List . . . 57
9.2 TypeList . . . 64
9.3 pipeline . . . 83
9.4 Compose . . . 91
9.5 延迟估值 . . . 92
9.6 Optional . . . 93

CHAPTER 1

引用

引用是 *C* 语言所没有的概念。而这个概念，比它表面看起来要复杂一些。

1.1 值与对象

为了理解引用，我们需要首先搞清楚什么叫 **左值**与 **右值**。

简而言之，**左值**是一种 **对象**，而不是 **值**。其关键区别在于，是否明确在内存中有其可访问的位置。即，其是否存在一个可访问的地址。如果有，那么它就是一个 **对象**，也就是一个 **左值**，否则，它就只是一个 **值**，即 **右值**。

比如：你不可能对整数 `10` 取地址，因而这个表达式是一个 **右值**。但是如果你定义了一个变量：

```
int a = 10;
```

变量 `a` 则代表一个 **对象**，即 **左值**。如果我们再进一步，表达式 `a + 1` 则是一个右值表达式，因为你无法对这个表达式取地址。

任何可以取地址的表达式，背后都必然存在一个 **对象**，因而也必然属于 **左值**。而如果我们把对象地址看作其 **身份证**（*Identifier*），那么我们也可以换句话说：任何有 **身份证**的表达式，都属于 **左值**；否则，肯定属于 **右值**。

1.2 别名

引用是 **对象**的 **别名**。

所谓 **别名**，是指你没有创建任何 **新事物**，而只是对 **已存在事物**赋予了另外一个名字。比如：

```
using Int = int;
```

你并没有创建一个叫做 `Int` 的新类型，而只是对已存在类型 `int` 赋予了另外一个名字。再比如：

```
template <typename T>
using ArrayType = Array<T, 10>;
```

你并没有创建一个签名为 `ArrayType<T>` 的新模版，而只是对已存在模版 `Array<T,N>` 进行部分实例化后得到的模版，赋予了一个新名字。

因而，**引用**作为 **对象别名**，并没有创建任何 **新对象**（包括引用自身），而仅仅是给已存在对象赋予了一个新名字。

1.3 空间

正是因为其 **别名**语义，*C++* 没有规定 **引用**的尺寸（事实上，从 **别名**语义的角度，它本身不需要内存，因而也就没有尺寸而言）。

因而，如果你试图通过 `sizeof` 去获取一个 **引用**的大小，是不可能的。你只能得到它所引用的对象的大小（由于别名语义）。

```
struct Foo {
  std::size_t a;
  std::size_t b;
};

Foo foo;
Foo& ref = foo;

static_assert(sizeof(ref) == sizeof(Foo));
```

也正是由于其 **别名语义**，当你试图对一个引用取地址时，你得到的是对象的地址。比如，在上面的例子中，`&ref` 与 `&foo` 得到的结果是一样的。

因而，当你定义一个指针时，指针自身就是一个 **对象**（左值）；它本身有自己明确的存储，并可以取自己的地址，可以通过 `sizeof` 获取自己的尺寸。

但是 **引用**，本身不是一个像指针那样的额外对象，而是一个对象的别名，**你对引用进行的任何操作，都是其所绑定对象的操作**。

在上面的例子中，ref 与 foo 没有任何差别，都是对象的一个名字而已。它们本身都代表一个对象，都是一个左值表达式。

因而，在不必要时，编译器完全不需要为引用分配任何内存。

但是，当你需要在一个数据结构中保存一个引用，或者需要传递一个引用时，你事实上是在存储或传递对象的 **身份**（即地址）。

虽然这并不意味着 sizeof(T&) 就是引用的大小（从语义上，引用自身非对象，因而无大小，sizeof(T&) == sizeof(T)），但对象的地址的确需要对应的空间来存储。

```
struct Bar {
   Foo& foo;
};

// still, reference keeps its semantics.
static_assert(sizeof(Bar::foo) == sizeof(Foo));

// but its storage size is identical to a pointer
static_assert(sizeof(Bar) == sizeof(void*));

// interesting!!!
static_assert(sizeof(Bar) < sizeof(Bar::foo));
```

1.4 受限的指针

在传递或需要存储时，一个引用的事实空间开销与指针无异。因而，在这些场景下，它经常被看作一个受限的指针：

1. 一个引用必须初始化。这是因为其 **对象别名**语义，因而没有 **绑定**到任何对象的引用，从语义上就不成立。

2. 由于必须通过初始化将引用绑定到某一个对象，因而从语义上，不存在 **空引用**的概念。这样的语义，对于我们的接口设计，有着很好的帮助：如果一个参数，从约束上就不可能是空，那么就不要使用指针，而使用引用。这不仅可以让被调用方避免不必要的空指针判断；更重要的是准确的约束表达。

   不过，需要特别注意的是：虽然 **空引用**从概念上是不存在的，但从事实上是可构造的。比如：T& ref = *(T*)nullptr 。

   因而，在项目中，任何时候，需要从指针转为引用时，都需要确保指针的非空性。

   另外，**空引用**本身这个概念就是不符合语义的，因为引用只是一个对象的别名。上面的表达式，事实上站在对象角度同样可以构造: T obj = *(T*)nullptr 。正如我们将指针所指向的对象赋值（或者初始化）给另一个对象一样，我们都必须确保指针的非空性。

3. 像所有的左值一样，引用可以绑定到一个抽象类型，或者不完备类型（而右值是不可能的）。从这一点上，指针和引用具有相同的性质。因而，在传递参数时，决定使用指针，还是引用，仅仅受是否允许为

空的设计约束。

4. 一个引用不可能从一个对象，绑定到 **另外**一个对象。原因很简单，依然由于其 **对象别名**语义。它本身就代表它所绑定的对象，重新绑定另外一个对象，从概念上不通。

   而引用的 **不可更换性**，导致任何存在引用类型非静态成员的对象，都不可能直接实现 **拷贝/移动赋值**函数。因而，标准库中，需要存储数据的，比如 **容器**，`tuple`，`pair`，`optional` 等等结构，都不允许存储 **引用**。

   这就会导致，当一个对象需要选择是通过 **指针**还是 **引用**来作为数据成员时，除了 **非空性**之外，相对于参数传递，还多了一个约束：**可修改性**。而这两个约束并不必然是一致的，甚至可以是冲突的。

   比如，一个类的设计约束是，它必须引用另外一个对象（非空性），但是随后可以修改为引用另外一个对象。这种情况下，使用指针就是唯一的选择。但代价是，必须通过其它手段来保证 **非空性**约束。

1.5 左值

任何一个引用类型的 **变量**，都必然是其所绑定 **对象**的 **别名**，因而都必然是 **左值**。无论这个引用类型是 **左值引用**，还是 **右值引用**。关于这个话题，我们会在后续章节继续讨论。

重要:

1. 引用是对象的别名，对于引用的一切操作都是对对象的操作；
2. 引用自身从概念上没有大小（或者就是对象的大小）；但引用在传递或需要存储时，其传递或存储的大小为地址的大小。
3. 引用必须初始化；
4. 引用不可能重新绑定；
5. 将指针所指向的对象绑定到一个引用时，需要确保指针非空。
6. 任何引用类型的变量，都是左值。

CHAPTER 2

右值引用

2.1 缺失的拼图

在 *C++11* 之前，表达式分类为 **左值表达式**和 **右值表达式**，简称 **左值**和 **右值**。**左值**都对应着一个明确的对象；从而也都必然可以通过 `&` 进行取地址操作。而 **右值**表达式虽然肯定都不能进行取地址操作，但在有些场景下，也会隐含着创建一个 **临时对象**的语意。

比如 `Foo(10)`，在 *C++98* 的年代，其语意是：以 `10` 来构造一个 `Foo` 类型的临时对象。而这个表达式属于 **右值**。

而引用，从 constness 的角度，可以分为：**non-const reference** 和 **const reference** 。

因而，**constness** 和 **引用**的 **对象类别**组合在一起，一共能产生四种类型的引用：

1. **const lvalue reference**
2. **non-const lvalue reference**
3. **const rvalue reference**
4. **non-const rvalue reference**

在 *C++11* 之前，通过符合 `&` 和 `const` 的两种组合，可以覆盖三种场景：

1. `Foo&`

- **non-const lvalue reference**

  比如：`Foo foo(10); Foo& ref = foo;`

2. `const Foo&`

- **const lvalue reference**

  比如：`Foo foo(10); const Foo& ref = foo;`

- **const rvalue reference**

  比如：`const Foo& ref = Foo(10);`

但对于 **non-const rvalue reference** 无法表达。

好在那时候并没有 `move` 语意的支持，因而对于 **non-const rvalue reference** 的需求也并不强烈。

2.2 move 语意

C++11 之前，只有 `copy` 语意，这对于极度关注性能的语言而言是一个重大的缺失。那时候程序员为了避免性能损失，只好采取规避的方式。比如:

```
std::string str = s1;
s += s2;
```

这种写法就可以规避不必要的拷贝。而更加直观的写法：

```
std::string str = s1 + s2;
```

则必须忍受一个 `s1 + s2` 所导致的中间 **临时对象**到 *str* 的拷贝开销。即便那个中间临时对象随着表达式的结束，会被销毁（更糟的是，销毁所伴随的资源释放，也是一种性能开销）。

对于 `move` 语意的急迫需求，到了 *C++11* 终于被引入。其直接的驱动力很简单：在构造或者赋值时，如果等号右侧是一个中间临时对象，应直接将其占用的资源直接 `move` 过来（对方就没有了）。

但问题是，如何让一个构造函数，或者赋值操作重载函数能够识别出来这是一个临时变量?

在 `C++11` 之前，拷贝构造和赋值重载的原型如下：

```
struct Foo {
   Foo(const Foo&);
   Foo& operator=(const Foo&);
};
```

参数类型都是 `const &`，它可以匹配到三种情况：

1. **non-const lvalue reference**
2. **const lvalue reference**
3. **const rvalue reference**

对于 **non-const rvalue reference** 是无能为力的。另外，即便是能捕捉 **const rvalue reference**，比如：`foo = Foo(10);`，但其 `const` 修饰也保证了其资源不可能被 `move` 走。

因而，能够被 `move` 走资源的，恰恰是之前缺失的那种引用类型：**non-const rvalue reference** 。

这时候，就需要有一种表示法，明确识别出那是一种 **non-const rvalue reference** ，最后定下来的表示法是 `T&&` 。这样，就可以这样来定义不同方式的构造和赋值操作：

```
struct Foo {
   Foo(const Foo&);    // copy ctor
   Foo(Foo&&);         // move ctor

   Foo& operator=(const Foo&); // copy assignment
   Foo& operator=(Foo&&);      // move assignment
};
```

通过这样的方式，让 `Foo foo = Foo(10)` 或 `foo = Foo(10)` 这样的表达式，都可以匹配到 `move` 语意的版本。与此同时，让 `Foo foo = foo1` 或 `foo = foo1` 这样的表达式，依然使用 `copy` 语意的版本。

2.3 右值引用变量

引入了 **右值引用**之后，就有一系列的问题需要明确。

首先，在不存在重载的情况下：

1. **左值**是否可以匹配到 **右值引用类型参数**？比如：

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

答案显然是 **NO** ，否则，一个左值就会被 `move ctor` 将其资源偷走，而这很明显不是我们所期望的；

2. **右值**是否可以匹配到 **左值引用类型参数**？比如：

```
struct non_movable {
   non_movable(const non_movable&);
};

struct non_movable2 {
   non_movable2(non_movable&);
};
```

答案是看情况。

- 至少在 *C++11* 之前，一个右值，就可以被类型为 `const T&` 类型的参数匹配；
- 但一个右值，不能被 `T&` 类型的参数匹配；毕竟这种可以修改的承诺。而修改一个调用后即消失的 **临时对象**上，没有任何意义，反而会导致程序员犯下潜在的错误，因而还是禁止了最好。

这就遗留下来一种情况：

3. 一个 **non-const rvalue reference** 类型的变量，是否允许匹配 **non-const lvalue reference** 类型的参数？

比如：

```
void f(Foo& foo) { foo.a *= 10; }

Foo&& ref = Foo{10};

f(ref); // 是否允许

int b = ref.a + 10;
```

没有任何理由不允许这样的匹配。毕竟，自从变量 `ref` 被初始化后，其性质上和 **左值引用**一样，都是引用了一个已经存在的对象。例子中，经过 `f(ref)` 对 `ref` 所引用的对象内容进行修改之后，还会基于其内容进行进一步的处理。这都是非常合理的需求。并且，`ref` 所引用的对象的生命周期，和 `ref` 一样长，不用担心在使用 `ref` 期间，对象已经不存在的问题。

这就导致了一个看起来很矛盾的现象：

```
void f(Foo& foo) { foo.a *= 10; }

Foo&& ref = Foo{10};
f(ref);      // OK

f(Foo{10}); // 不允许
```

先将一个 **临时对象**初始化给一个 **右值引用**，再传递给函数 `f`，与直接构造一个 **临时对象**传递给 `f`，一个是允许的，一个是禁止的。

这背后的差异究竟意味这什么？

一个类型为 **右值引用**的变量，一旦被初始化之后，临时对象的生命将被扩展，会在其被创建的 *scope* 内始终有效。因而，`Foo&& foo = Foo{10}`，从语意上相当于：

```
{
   Foo __temp_obj{10};
   Foo& ref = __temp_obj;

   // 各种对ref的操作
}
// 离开scope, __temp_obj被销毁
```

因而，看似 `foo` 被定义的类型为 **右值引用**，但这 **仅仅约束它的初始化**：只能从一个 **右值**进行初始化。但一旦初始化完成，它就和一个 **左值引用**再也没有任何差别：都是一个已存在对象的 **标识**。

函数参数也没有任何特别之处，它就是一个普通的变量。无非是其可访问范围被限定在函数内部。调用一个

函数时，传递实参的过程，就是一个对参数（变量）进行初始化的过程，而初始化的细节与一个普通变量没有任何差别。

```
void stupid(Foo&& foo) {
   foo.a += 10;   // 在函数体内，foo的性质与一个左值引用毫无差别
   // blah ...
}

stupid(Foo{10});  // 在执行函数体之前，进行参数初始化：Foo&& foo = Foo{10}
```

而临时对象 `Foo{10}` 的生命周期，会比参数变量 `foo` 更长。所以将 `foo` 看作 **左值引用**随意访问，是没有任何风险的。

所以，任何一个类型为 **右值引用**的变量，一旦初始化完成，性质上就变成和一个 **左值引用**毫无差别。这样的语意，对于程序员的使用是最为合理的。

我们再看下面的例子：

```
std::string&& ref = std::string("abc");

std::string obj = ref; // move? 还是 copy?

std::string s = ref + "cde"; // 是否可以接着假设ref所引用的对象是合法的？
```

既然在完成初始化之后，一个 **右值引用类型**的变量，就变成了 **左值引用**，按照这个语意，当然就只能选择 `copy` 构造。这样的选择，也让后面对于 `ref` 的继续使用是安全合理的，这其实也在帮助程序员编写安全的代码。

毕竟，只有在调用 `move constructor` 那一刻，传入的是真正的临时变量，也就是说 `move constructor` 调用结束后，临时变量也就不再存在，无从访问的情况下，自动选择 `move constructor` 才是确定安全的。

经过之前讨论，我们知道这样的设计决策是最合理的，但矛盾和张力依然存在：毕竟，变量 `ref` 的类型是 **右值引用**，而 `move constructor` 的参数类型也是 **右值引用**，为什么它们不是最匹配的，反而是匹配了 `copy constructor`？另外，`move constructor` 自动匹配真正的临时对象，毫无疑问是合理的（也是我们的初衷），但我们如何区分一个临时对象和一个类型为 **右值引用**的变量?

这个并不难。因为 C++ 早就规定了，产生临时变量的表达式是 **右值**，而任何变量都是一个对象的标识，因而都是 **左值**，哪怕变量类型是 **右值引用**。

因而，**右值**选择 `move constructor`，**左值**选择 `copy constructor`。

更准确的说，所谓选择 `move constructor`，其实是因为 **右值**匹配的是 `move constructor` 参数，其类型是一个 **右值引用**。我们知道，函数参数也是变量，而一个类型为 **右值引用**的变量，只能由 **右值**来初始化：

```
Foo   foo{10};
Foo&& ref = foo; // 不合法，右值引用只能由右值初始化
```

(下页继续)

(续上页)

```
Foo&& ref1 = Foo{10};
Foo&& ref2 = ref1; // 不合法，ref1是个左值
```

因而，做为类型为 **右值引用**的函数参数，唯一能匹配的就是 **右值**。这也是 `move constructor` 能精确识别临时变量的原因。

重要:

1. 对于任何类型为 **右值引用**的变量（当然也包括函数参数），只能由 **右值**来初始化；
2. 一旦初始化完成，**右值引用**类型的变量，其性质与一个 **左值引用**再也没有任何差别。

2.4 速亡值

我们现在已经明确了，只有右值临时对象可以初始化右值引用变量，从而也只有右值临时变量能够匹配参数类型为 **右值引用**的函数，包括 `move` 构造函数。

这中间依然有一个重要的缺口：如果程序员就是想把一个左值 `move` 给另外一个对象，该怎么办?

最简单的选择是通过 `static_cast` 进行类型转换：

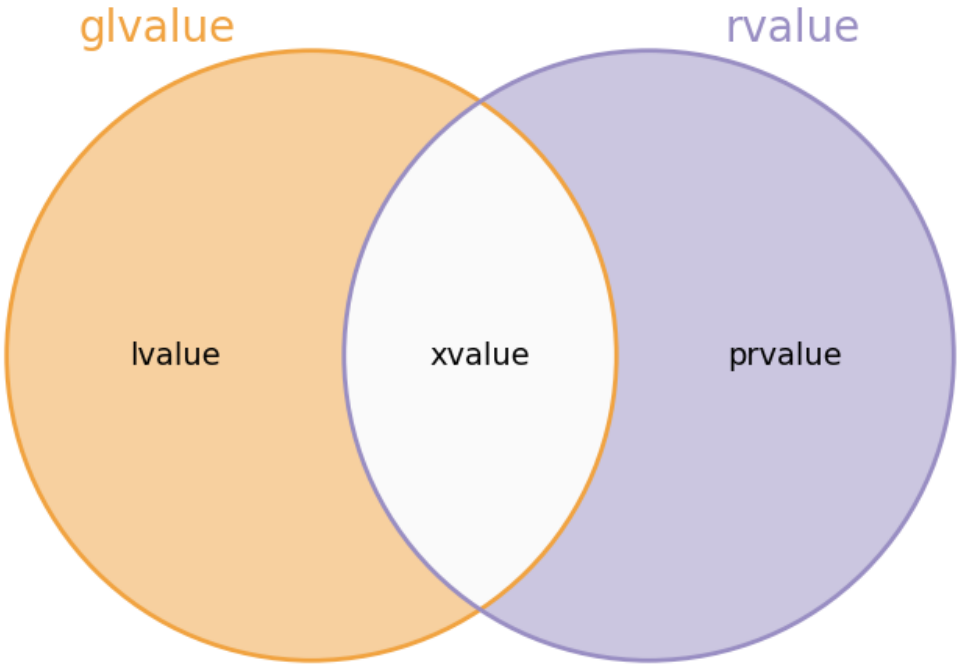
```
Foo   foo{10};
Foo&& ref = Foo{10};

Foo obj1 = static_cast<Foo&&>(foo); // move 构造
Foo obj2 = static_cast<Foo&&>(ref); // move 构造
```

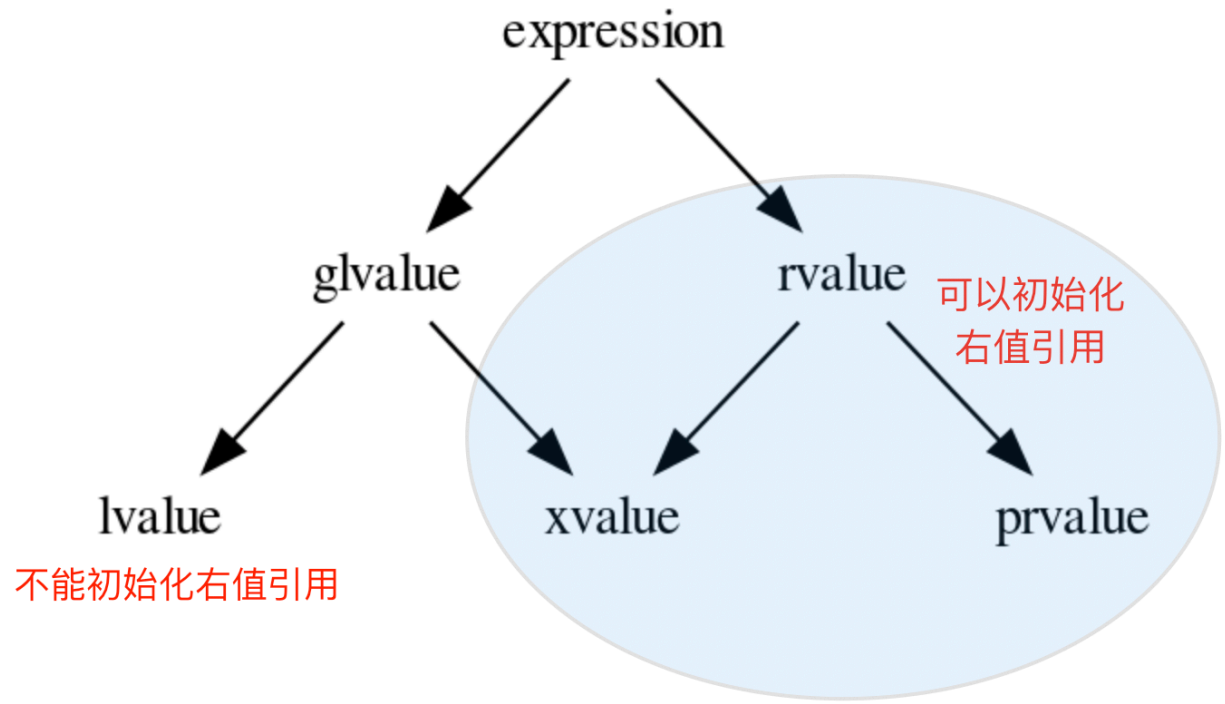
我们之前说过，只有 **右值**，才可以用来初始化一个 **右值引用**类型的变量，因而也只有 **右值**才能匹配 `move` 构造。所以，`static_cast<Foo&&>(foo)` 表达式，肯定是一个 **右值**。

但同时，它返回的类型又非常明确的是一个 **引用**，而这一点又不符合 **右值**的定义。因为，所有的右值，都必须是一个 **具体类型**，不能是不完备类型，也不能是抽象类型，但 **引用**，无论左值引用，还是右值引用，都可以是不完备类型的引用或抽象类型的引用。这是 **左值**才有的特征。

对于这种既有左值特征，又和右值临时对象一样，可以用来初始化右值引用类型的变量的表达式，只能将其归为新的类别。*C++11* 给这个新类别命名为 **速亡值** (eXpiring value，简称 xvalue)。而将原来的 **右值**，重新命名为 **纯右值**。而 **速亡值**和 **纯右值**合在一起，称为 **右值**，其代表的含义是，所有可以直接用来初始化 **右值引用类型变量**的表达式。

同时，由于 **速亡值**又具备左值特征：可以是不完备类型，可以是抽象类型，可以进行运行时多态。所以，**速亡值**又和 **左值**一起被归类为 **泛左值**（generalized lvalue, 简称 glvalue)。


glvalue
rvalue
lvalue
xvalue
prvalue



除了 `static_cast<T&&>(expr)` 这样的表达式之外，任何返回值为 **右值引用**类型的函数调用表达式也属于 **速亡值**。从而让用户可以实现任意复杂的逻辑，然后通过返回值为 **右值引用**的方式，直接初始化一个右值引用类型的变量。以此来达到匹配 `move` 构造，`move` 赋值函数，以及任何其它参数类型为 **右值引用**的函数的目的。

C++ 标准对其的定义为：

xvalue: an xvalue (an "eXpiring" value) is a glvalue that denotes an object or bit-field whose resources can be reused.

意思就是，这类表达式表明了自己可以被赋值给一个类型为 **右值引用**的变量，当然自然也就可以被 `move` 构造和 `move` 赋值操作自然匹配，从而返回的引用所引用的对象可以通过 `move` 而被重用。

所以，速亡值未必真的会速亡（expiring），它只是能用来初始化右值引用类型的变量而已。只有用到 `move` 场景下，它才会真的导致所引用对象的失效。

最后，速亡表达式存在着一个异常场景，那就是函数类型的右值引用。因为函数地址被 `move` 本身毫无意义。所以，对于返回值为 **函数类型右值引用**的函数调用，或者 `static_cast<FunctionType&&>(expr)` 的表达式，其类别为 **左值**，而不是 **速亡值**。

重要:

- 类型为 **右值引用**的变量，只能由 **右值**表达式初始化；
- **右值**包括 **纯右值**和 **速亡值**，其中 **速亡值**的类型是 **右值引用**；

- 类型为 **右值引用**的变量，是一个 **左值**，因而不能赋值给其它类型为 **右值引用**的变量，当然也不能匹配参数类型为 **右值引用**的函数。

CHAPTER 3

值与对象

在理解 *Modern C++* 的各种令人眼花缭乱的特性之前，必须先搞清楚两个基本概念：**对象**（*object*）和 **值**（*value*）。这是理解很多特性的基础。

3.1 值

简单说，**值**是一个纯粹的数学抽象概念，比如数字 `10`，或者字符 `'a'`，或者布尔值 `false`，等等。它们完全不需要依赖于计算机或者内存而存在，就只是一个纯粹的值：不需要存储到内存，当然也就不可修改。注意，这与存储在内存中，但 *immutable* 完全不是一个语意。

那么 `1+2` 呢？这是一个表达式，但这个表达式的求值结果也是一个 **值**。因而，这是一个值类别的表达式。而数字 `10` 同样是一个表达式，其求值的结果毫无疑问也是一个 **值**——它自身。因而，在这个角度，`1+2` 和数字 `10`，从性质上没有任何区别，都是 **值**类别的表达式。

3.2 对象

对象是一个在内存中占据了一定空间的有类型的东西。因而，它必然是与计算机内存这个物理上具体存在的设备关联在一起的一个物质。

因而，每一个对象都必然有一个 **标识**（*Identifier*），从而你可以知道这个对象在内存中唯一的起始位置。否则，对象是一个与内存关联在一起的物质就无从谈起。

所以 `int i` 就定义了一个对象，系统必然会在内存中为其分配一段 `sizeof(int)` 大小的空间，而 `i` 就是这个对象的标识。

既然对象与内存有关联，并且有自己区别于其它对象的唯一起始内存地址，那么任何对象都必然可以被引用。引用做为一个对象的别名，当然也是对象的一种 **标识**。

所以，区分 **对象**和 **值**的方法非常简单：是否有 **标识**，或可否被 **引用**（毕竟引用就是一种标识）。只有做为具体内存物质的对象才可能被引用；而值，做为一种抽象概念，引用无从谈起。

3.3 值与对象的关系

那么 **值**和 **对象**之间是什么关系?

很简单，**值**用来初始化 **对象**。比如：`bool b = true`，其语意是：用值 `true` 初始化对象 `b`；类似的，`int i = 1 + 2` 表示用值 `1+2` 的计算结果值，初始化对象 `i`。**对象**表示内存中的一段有类型的空间，**值**这则是个空间里的内容。用 **值**来初始化 **对象**的过程，是一个将值加载到空间的隐喻。

3.4 纯右值

所有的 **值**语意的表达式，都归类为 **纯右值**（*pure right value*，简称 *prvalue*）。在 *C++11* 之前，它们被称做 **右值**。

规范对于纯右值的定义如下：

> A *prvalue* is an expression whose evaluation initializes an object or a bit-field, or computes the value of an operand of an operator, as specified by the context in which it appears, or an expression that has type cv void.

其存在的唯一的目的，是为了初始化 **对象**。单独写一个 **纯右值**表达式的语句，比如：`1+2;`，或者 `true && (1 == 2);`，这样的表达式被称做 *弃值表达式*。从语意上，它们仍然会初始化一个临时对象，而临时对象也是泛左值。后面我们会进行解释。

而既然是一个 **值**，就必须是某种具体类型的值，而不可能是某种 **不完备类型**。当然也不可能是一个 **抽象类型**（包含有纯虚函数的类）的值，即便其基类是某种抽象类型，但它自身必然是一个具体类型，因而对其任何 `virtual` 函数的调用，都是对其具体类型所对应的函数实现的调用。

同时，你不可能对一个值进行取地址操作（语意上就不通），也不可能引用它。

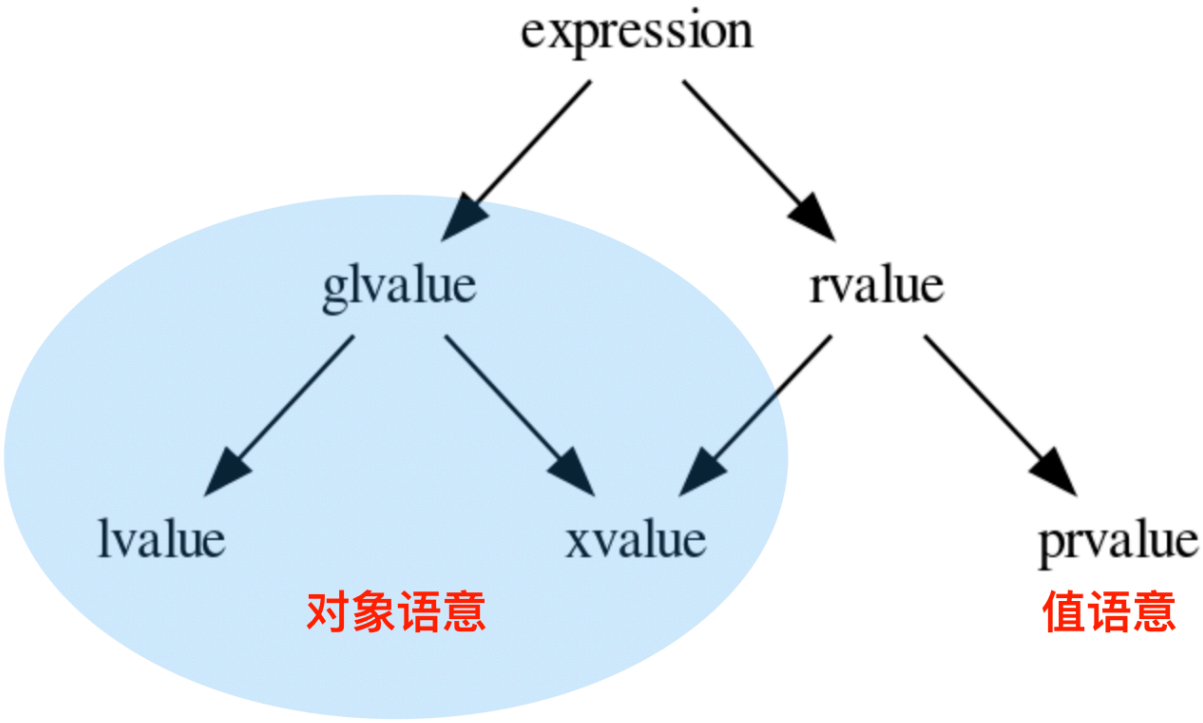
3.5 泛左值

与 **纯右值**对应的是 **泛左值**（*glvalue*）。整个表达式的世界被分为这两大类别。前者全部是 **值**语意，后者全部是 **对象**语意。

规范对于泛左值的定义如下：

> A glvalue is an expression whose evaluation determines the identity of an object, bit-field, or function.

从这个定义我们可以看出，泛左值表达式的求值结果是一个对象的标识。



3.5.1 左值

左值很容易辨别：任何可以对其通过符号 `&` 取地址的表达式，都属于 **左值**。因而，任何变量（包括常量），无论是全局的，还是类成员的，还是函数参数，还是函数名字，都肯定属于左值。

另外，所有返回值是左值引用的函数调用表达式（包括用户自定义的重载操作符），以及 `static_cast<T&>(expr)` 都必然也属于左值。毕竟，没有内存中的对象，哪里来的引用？而引用无非是对象的一个别名标识罢了。

剩下的就是系统的一些 *builtin* 操作符的定义，比如对一个指针求引用操作：`*p`，或者 `++i`（`i++` 却是一个 **右值**）。

其中，最为特殊的是字符串字面常量，比如：`"abcd"`，这是一个左值对象。这有点违背直觉，但由于 *C/C++* 中字符串并不是一个 *builtin* 基本类型。这些字符串字面常量都会在内存中得以存储。

需要注意的是，这两种情况下，无论是变量 `i`，还是函数参数 `r`，它们都是一个 **左值**，虽然它们的类型是 **右值引用**。我们之前谈到过，任何变量，无论其属于什么类型，都必然是一个左值。变量的名字，就是对应对象的标识。

3.5.2 速亡值

速亡值是所有返回类型为 **右值引用**的非左值表达式。这包括返回值类型为 **右值引用**的函数调用表达式，`static_cast<T&&>(expr)` 表达式。

其所引用的对象，从理论上同样也是可以取其地址的。其目的是为了初始化类型为 **右值引用**类型的变量。借此，也可以匹配参数类型为右值引用的函数。一旦允许取其地址，程序的其它部分将无从判断，一个地址来自于速亡值对象，还是来自于左值对象，从而让速亡值的存在失去了本来的意义。因而，对其取地址操作被强行禁止。

与右值引用和速亡值有关的详细讨论，请参考：*右值引用* 。

3.5.3 对象？值？

上面给的那些与值有关的例子，简单而直观，不难理解它们是数学意义上的值。我们来看一个不那么直观的例子：在 `Foo` 是一个 `class` 的情况下，`Foo{10}` 是一个对象还是一个值?

在 *C++17* 之前，这个表达式的语意是一个 **临时对象**。

非常有说服力的例子是：`Foo&& foo = Foo{10}` 或者 `const Foo& foo = Foo{10}` 。这这两个初始化表达式里，毫无疑问 `Foo{10}` 是一个对象，因为它可以被引用，无论是一个右值引用 `Foo&&` ，还是一个左值引用 `const Foo&`，能被引用的必然是 对象。

但后来人们发现，将其定义为对象语意，在一些场景下会带来不必要的麻烦：

比如：`Foo foo = Foo{10}` 的语意是：构造一个临时对象，然后 `copy/move` 给左边的对象 `foo` 。

注意，只要 `Foo{10}` 被定义为 **对象**，那么 `copy/move` 语意也就变得不可避免，这就要求 `class Foo` 必须要隐式或显式的提供 `public copy/move constructor` 。即便编译器肯定会将对 `copy/move constructor` 的调用给优化掉，但这是到优化阶段的事，而语意检查发生在优化之前。如果 `class Foo` 没有 `public copy/move constructor` ，语意检查阶段就会失败。

这就给一些设计带来了麻烦，比如，程序员不希望 `class Foo` 可以被 `copy/move` ，所有 `Foo` 实例的创建都必须通过一个工厂函数，比如：`Foo makeFoo()` 来创建；并且程序员也知道 `copy/move constructor` 的调用必然会被任何像样的编译器给优化掉，但就是过不了那该死的对实际运行毫无影响的语意检查那一关。

于是，到了 *C++17* ，对于类似于 `Foo{10}` 表达式的语意进行了重新定义，它们不再是一个 **对象**语意，而只是一个 **值**。即 `Foo{10}` 与内存临时对象再无任何关系，它就是一个 **值**：其估值结果，是对构造函数 `Foo(int)` 进行调用所产生的 **值**。而这个 **值**，通过等号表达式，赋值给左边的 **对象**，正如 `int i = 10` 所做的那样。从语意上，不再有对象间的 `copy/move` ，而是直接将构造函数调用表达式作用于等号左边的 **对象**，从而完成用 **值**初始化 **对象**的过程。因而，`Foo foo = Foo{10}` ，与 `Foo foo{10}` ，在 *C++17* 之后，从语意上（而不是编译器优化上）完全等价。

一旦将其当作 **值**语意，很多表达式的理解上也不再一样。比如：`Foo foo = Foo{Foo{Foo{10}}}` ，如果 `Foo foo = Foo{10}` 与 `Foo foo{10}` 完全等价，那么就可以进行下列等价转换：

```
    Foo foo = Foo{Foo{Foo{10}}}
<=> Foo foo{Foo{Foo{10}}}
<=> Foo foo = Foo{Foo{10}}
<=> Foo Foo{Foo{10}}
<=> Foo foo = Foo{10}
<=> Foo foo{10}
```

注意，这是一个自然的语意推论，而不是编译器优化的结果。

自然，对于 `Foo makeFoo()` 这样的函数，其调用表达式 `makeFoo()`，在 *C++17* 下也是 **值**。而不像之前定义的那样：返回一个临时对象，然后在 `Foo foo = makeFoo()` 表示式里，copy/move 给等号左侧的对象 `Foo`。虽然 *C/C++* 编译器很早就有 RVO/NRVO 优化技术；但同样，那是优化阶段的事，而不是语意分析阶段如何理解这个表达式语意的问题。

3.5.4 纯右值物质化

我们再回到前面的问题：`Foo&& foo = Foo{10}` 表达了什么语意？毕竟，按照我们之前的讨论，等号右边是一个 **值**，而左边是一个对于对象的 **引用**。而 **引用**只能引用一个对象，引用一个 **值**是逻辑上是讲不通的。

这中间隐含着一个过程：**纯右值物质化**。即将一个 **纯右值**，赋值给一个 **临时对象**，其标识是一个无名字的 **右值引用**，即 **速亡值**。然后再将等号左边的 **引用**绑定到这个 **速亡值**对象上。

纯右值物质化的过程还发生在其它场景。

比如，`Foo{10}` 是一个 **纯右值**，但如果我们试图访问其非静态成员，比如：`Foo{10}.m`，此时就必需要将这个纯右值物质化，转化成 **速亡值**。毕竟，对于任何非静态成员的访问，都需要对象的 **地址**，与成员变量所代表的 **偏移**两部分配合。没有对象的存在，仅靠偏移量访问其成员，根本不可能。

还有数组的订阅场景。比如:

```
using Array = char [10];

Array{};    // 纯右值
Array{}[0]; // 速亡值
```

另外，`static_cast<T>(expr)` 是一个 **直接初始化**表达式，即，中间存在一个隐含的 `T` 类型的未命名临时变量，通过 `expr` 进行初始化。如果 `expr` 是一个 **纯右值**，而 `T` 是一个 **右值引用**类型，则这个过程也是一个纯右值 **物质化**的过程。

而之前提到的弃值表达式，也会有一个 **纯右值物质化**的过程。这样的表达式的存在主要是为了利用其副作用。如果编译器发现其并不存在副作用，往往会将其优化掉。但这是优化阶段的职责。在语意分析阶段，统统是 **纯右值物质化**语意。

在 *C++17* 之前的规范定义中，将 **纯右值**和 **速亡值**合在一起，称为 **右值**。代表它们可以被一个 **右值引用类型的变量**绑定（即初始化一个右值引用类型的变量）。因而，在进行重载匹配时，**右值**会优先匹配 **右值引用类型的参数**。比如:

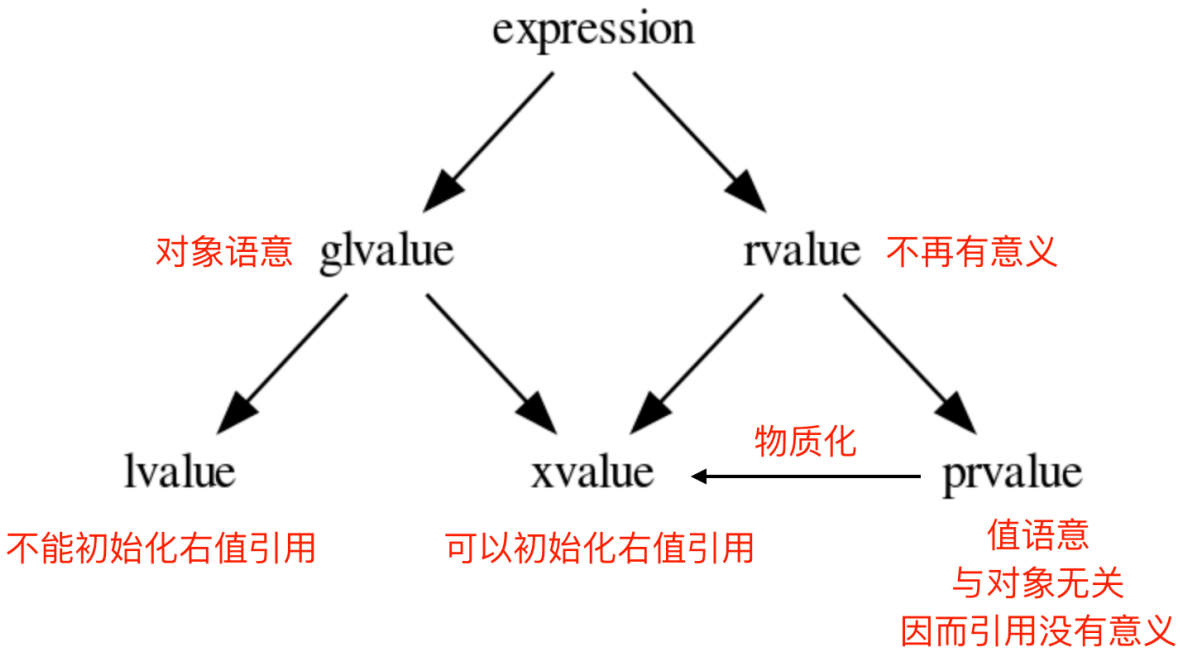
```
void func(Foo&&);        // #1
void func(const Foo&);   // #2

Foo&& f();



func(Foo{10}); // #1
func(f());     // #1

Foo foo{10};
func(foo);     // #2

Foo&& foo1 = Foo{10};
func(foo1);    // #2
```

到了 *C++17*，从匹配行为上没有变化，但语意上却有了变化。最终导致匹配右值引用版本的不是 **纯右值**类别，而是 **速亡值**。因为 **纯右值**会首先进行 **物质化**，得到一个 **速亡值**。最终是用 **速亡值**初始化了函数的对应参数。

一个 **纯右值**，永远也无法匹配到 `move` 构造函数。因为 `Foo foo = Foo{10}` 与 `Foo foo{10}` 等价。这不需要将 **纯右值**进行 **物质化**，得到一个 **速亡值**，然后匹配到 `move` 构造函数的过程。

只有 **速亡值**，才能匹配到 `move` 构造。比如：`Foo foo = std::move(Foo{10})` 将会导致 `move` 构造的调用。

另外，一个表达式是 **速亡值**，并不代表其所引用的对象一定是一个从 **纯右值** **物质化**得到的临时对象。而是两种可能都存在。比如，如果 `foo` 是一个 **左值**，`std::move(foo)` 这个 **速亡值**所引用的对象就是一个 **左值**；而 `std::move(Foo{10})` 则毫无疑问引用的是一个 **物质化**后的到的临时对象。



注意:

- 所有的表达式都可以归类为 **纯右值**和 **泛左值**；
- 所有的 **纯右值**都是 **值**的概念；所有的 **泛左值**都是 **对象**的概念；
- **左值**可以求地址，**速亡值**不可以求地址；
- **纯右值**在需要临时对象存在的场景下，会通过 **物质化**，转化成 **速亡值**。
- **泛左值**可以是抽象类型和不完备类型，可以进行多态调用；**纯右值**只能是具体类型，无法进行多态调用。
- 用 **纯右值**构造一个 **左值**对象时，是 **直接构造**语意；用 **速亡值**构造一个 **左值**对象时，是 **拷贝/移动构造**语意。

CHAPTER 4

decltype

decltype 是 *C++11* 加入的一个重要特性。它允许求一切合法表达式的类型。从而，让从类型到值，从值到类型形成了一个闭环，极大的扩展了泛型编程的能力。

C++ 规范中，对于 decltype 类型推演规则的定义如下：

1. 若实参为无括号的 **标识表达式**或无括号的 **类成员访问表达式**，则 decltype 产生以此表达式命名的实体的类型。若无这种实体，或该实参指名某个重载函数，则程序非良构。
2. 若实参是其他类型为 T 的任何表达式，且
   a) 若表达式的值类别为 **速亡值**，则 decltype 产生 T&& ；
   b) 若表达式的值类别为 **左值**，则 decltype 产生 T& ；
   c) 若表达式的值类别为 **纯右值**，则 decltype 产生 T 。

   若表达式是 **纯右值**，则不从该纯右值 **物质化**临时对象：这种纯右值无结果对象。

   注意如果对象的名字带有括号，则它被当做通常的 **左值**表达式，从而 decltype(x) 和 decltype((x)) 通常是不同的类型。

这些规则，初看起来，有些让人困惑。但如果真的理解了背后的机制，其实非常容易理解。

decltype 有两种表达方法：

1. 有括号：decltype((expr))
2. 无括号：decltype(expr)

4.1 有括号语意

有括号的表达方法，语意是简单而统一的：它站在表达式类别的角度求类型。

1. 如果表达式属于 **纯右值**，结果必然是 **非引用**类型；
2. 如果表达式属于 **泛左值**，结果必然是 **引用**类型；
    - 如果表达式属于 **左值**，结果必然是 **左值引用**；
    - 如果表达式属于 **速亡值**，结果必然是 **右值引用**；

```
struct Foo { int a; };

using Func  = Foo& ();
using Array = char[2];

enum class E { OK, FAIL };

const Foo f_v();
Foo&      f_ref();
Foo&&     f_r();

int         a    = 0;
const int   b    = 1;
const Foo   foo  = {10};
Foo&&       rref = Foo{1};
const Foo&  ref  = foo;
char        c[2] = {1, 2};
int*        p    = &a;
const Foo* pFoo = &foo;

// 左值
decltype((a))        v1;   // int&
decltype((foo))      v2;   // const Foo&
decltype((foo.a))    v3;   // const int&
decltype((f_ref())) v4;   // Foo&
decltype((f_r))      v5;   // Foo&& (&)()
decltype((c))        v6;   // char (&)[2]
decltype((a += 10)) v7;   // int&
decltype((++a))      v8;   // int&
decltype((c[1]))     v9;   // char&
decltype((*p))       v10;  // int&
decltype((p))        v11;  // int*&
decltype((pFoo))     v12;  // const Foo*&
decltype((pFoo->a)) v13;  // const int&
```

(下页继续)

(续上页)

```
decltype((Foo::a))  v14;  // int&
decltype((rref))    v15;  // Foo&
decltype((ref))     v16;  // const Foo&
decltype((a > 0 ? a : b))              v;  // int&
decltype((static_cast<Func&&>(f_ref))) f;  // Foo& (&)()

// 纯右值
decltype((1+2))          v1;   // int
decltype((Foo{10}))      v2;   // Foo
decltype((f_v()))        v3;   // const Foo
decltype((Array{0, 1})) v4;   // char[2]
decltype((a++))          v5;   // int
decltype((&b))           v6;   // const int*
decltype((OK))           v7;   // E
decltype((a > 0 ? 10 : Foo{0}.a)) v;  // int

// 速亡值
decltype((Foo{10}.a))    v1;  // int&&
decltype((f_r()))        v2;  // Foo&&
decltype((Array{}[0]))   v3;  // char&&
decltype((std::move(a))) v4;  // int&&
decltype((a > 0 ? Foo{1}.a : Foo{0}.a)) v;  // int&&
```

这其中，最有趣的是 `decltype((rref))` ，`rref` 本身的类型是一个右值引用 `Foo&&` ，但做为左值表达式，它的类型却是 `Foo&` ，而这一点，请参见右值引用变量。

4.2 无括号语意

无括号的情况下，除了一种例外，其它情况下，都与有括号场景一致。

这个例外就是对于变量（包括常量）名字的直接求类型。这种情况，会返回变量被定义时的类型。

```
struct Foo { int a; };

using Func  = Foo& ();
using Array = char[2];

const Foo f_v();
Foo&      f_ref();
Foo&&     f_r();

int a           = 0;
const int  b    = 1;
```

(下页继续)

(续上页)

```
const Foo  foo  = {10};
Foo&&      rref = Foo{1};
const Foo& ref  = foo;
char       c[2] = {1, 2};
int*       p    = &a;
const Foo* pFoo = &foo;

decltype(a)         v1;   // int
decltype(b)         v2;   // const int
decltype(foo)       v3;   // const Foo
decltype(ref)       v4;   // const Foo&
decltype(rref)      v5;   // Foo&&
decltype(c)         v6;   // char[2]
decltype(p)         v7;   // int*
decltype(foo.a)     v8;   // int
decltype(ref.a)     v9;   // int
decltype(rref.a)    v10;  // int
decltype(pFoo)      v11;  // const Foo*
decltype(pFoo->a)   v12;  // int
decltype(Foo{1).a)  v13;  // int
decltype(Foo::a)    v14;  // int
```

从例子中不难看出，对于所有的变量访问，无论直接还是间接，由于每个变量在定义时都有自己的类型，因而求类型的结果就是这些变量被定义时的类型。

所以，之所以会出现有括号，无括号两种用法，正是因为每一个被定义的变量，都面临着两种需求：

1. 它们被定义时的类型
2. 整体做为一个表达式的类型（一定是泛左值）

前者是不关心表达式的，比如 `decltype(Foo{1}.a)` ，它只关心 `a` 被定义时的类型：`int` ；而不关心整个表达式本身是一个 `xvalue` ，因而表达式必然应该是一种右值引用类型：`int&&` 。

正是对于变量有这两种需求的存在，而其它表达式没有这样的问题，所以，才专门为变量定义了两种求类型的方法。而对于其它表达式则两种方式无差别。

CHAPTER 5

auto 类型推演

auto 类型推演脱胎于模版函数的类型推演，它们的能力几乎等价（除了初始化列表的情况）。这也就意味着，其实在 C++11 之前，C++ 早就具备了 auto 的能力，只是没有从语法上允许而已。

5.1 auto 的语意

和直觉不同的是，对于任意表达式：`auto v = expr`，`v` 的类型并不总是和 `expr` 所返回的类型一致。

首先，auto 不可能是一个 **引用**，无论是 **左值引用**，还是 **右值引用**，所以，如果 `expr` 返回类型里包含任何引用，都会被舍弃。比如：

```
Foo foo{1};
Foo& ref = foo;
Foo&& rref = Foo{2};

Foo& getRef();
Foo&& getRref();

auto v1 = ref;         // v1 type: Foo
auto v2 = rref;        // v2 type: Foo
auto v3 = getRef();    // v3 type: Foo
auto v4 = getRref();   // v4 type: Foo
```

其次，所有对值所修饰的 const 都会被丢弃。比如：

```
const Foo foo{1};
const Foo& ref     = foo;
const Foo&& rref   = Foo{2};
const Foo* const p = &foo;

auto v1 = foo;   // Foo
auto v2 = ref;   // Foo
auto v3 = rref;  // Foo
auto v4 = p;     // const Foo*
```

究其原因，是因为这种直接给出 auto 的写法，是一种 copy/move 语意。因而，等号右边的表达式本身类型是引用，并不影响等号左侧对象本身不是引用；同样的，等号右边表达式本身的 **constness** ， copy/move 后，并不会影响新定义变量的 **constness** 。

其推演语意，完全等价于：

```
template <typename T>
void f(T value);
```

其中 T 就是 auto ， value 就是你用 auto 所定义的变量。

注意，到了 C++17 之后，并非所有的场景下，都是 copy/move 语意，比如 auto v = Foo{1} ，其行为完全等价于： Foo v{1} 。具体可参见对象？值？ 。

因而，更准确的说，这不是 copy/move 语意，而属于构造初始化语意。

5.2 引用及 const

因而，如果你希望让新定义的变量属于引用类型，或具备 const ，则需要明确指定。比如：

```
auto        foo  = Foo{1};
const auto& ref  = foo;
auto&&      rref = Foo{2};
```

而指针的情况则稍有特殊。

5.3 指针

当你不指定指针的情况下，如果等号右侧的表达式是一个指针类型，那么左侧的变量类型当然也是一个指针。

当你明确指定指针的情况下，则是要求右侧表达式必须是一个指针类型。

```
Foo foo{1};
Foo* pFoo = &foo;

auto  v1 = foo;  // v1 type: Foo
auto  p1 = pFoo; // p1 type: Foo*
auto* p2 = pFoo; // p2 type: Foo*
auto* p3 = foo;  // Error: foo is not a pointer
```

5.4 通用引用

更为特殊的是 `auto&& v = expr` 的表达式。这并不必然导致 `v` 是一个右值引用。而是取决于 `expr` 的类别。

- 如果 `expr` 是一个 **左值**表达式，那么 `v` 将是左值引用类型；
- 如果 `expr` 是一个 **右值**表达式（参见对象？值？），那么 `v` 将会是右值引用类型。

```
1  Foo   foo{1};
2  Foo&  ref = foo;
3  Foo&& rref = Foo{2};
4  Foo&& getRref();
5  Foo&  getRef();
6  Foo   getFoo();
7
8  auto&& v1 = foo;            // v1 type: Foo&
9  auto&& v2 = Foo{2};         // v2 type: Foo&&
10 auto&& v3 = getRref();      // v3 type: Foo&&
11 auto&& v4 = getRef();       // v4 type: Foo&
12 auto&& v5 = getFoo();       // v5 type: Foo&&
13 auto&& v6 = ref;            // v6 type: Foo&
14 auto&& v7 = rref;           // v7 type: Foo&
```

正是因为这样的写法，允许等号右侧是任意合法的表达式，而等号左侧总是可以根据表达式类别，推演出合适的引用类型。所以这种写法被称做 **通用引用**。

其中，我们可以清晰的看出，虽然 `ref` 和 `rref` 分别被定义为 **左值引用**和 **右值引用**，但它们做为左值来讲，是等价的。都是左值引用。具体可参考右值引用变量 。

5.5 初始化列表

由于初始化列表不是一个表达式，因而类型也就无从谈起。所以 C++14 对其做了特殊的规定：

- 如果使用 **直接初始化**（不用等号）的方式，比如：`auto i{1}`，则初始化列表只允许有一个元素，其等价于 `auto i = 1`；如果初始化列表超过一个元素，比如 `auto j{1,2}`，则编译失败。
- 如果使用 **拷贝初始化**（用等号）的方式，比如：`auto v = {1, 2}`，则初始化列表允许有多个同一类型的元素。其等价于 `std::initializer_list<int> v = {1, 2}`。而 `auto v = {1}` 则等价于 `std::initializer_list<int> v = {1}`。

5.6 decltype(auto)

由于 `auto` 推演总是会丢弃 **引用**及 `const` 信息，明确给出 **引用**又总是得到一个引用。明确给出 `const`，则总是得到一个 `const` 类型。这对于想精确遵从等号后面类型的情况非常不便，尤其在进行泛型编程时，很难通过 *auto* 符合通用的情况。

而 `decltype` 恰恰相反，它总是能准确捕捉右侧表达式的类型（参见*decltype*）。因而，我们可以这样写：

```
Foo foo{1};
const Foo& ref = foo;
Foo&& rref = Foo{2};
int a = 0;

decltype(foo)     v1 = foo;   // Foo
decltype((foo))   v2 = foo;   // Foo&
decltype(ref)     v3 = ref;   // const Foo&
decltype(rref)    v4 = rref;  // Foo&&
decltype((rref))  v5 = rref;  // Foo&
decltype(1+2)     v6 = 1 + 2; // int

decltype((a > 0 ? Foo{0}.a : Foo{1}.a)) v7 = \
   a > 0 ? Foo{0}.a : Foo{1}.a; // int&&
```

但这样的写法，总是要把右边的表达式在 `decltype` 里重复写一遍，才能做到。到了 C++14，推出了一种新的写法：`decltype(auto)`，其中 `auto` 是一个自动占位符，代表等号右侧的表达式，这就大大简化了程序员的工作：

```
decltype(auto)   v1 = foo;    // Foo
decltype(auto)   v2 = (foo);  // Foo&
decltype(auto)   v7 = (a > 0 ? Foo{0}.a : Foo{1}.a); // int&&
```

5.7 函数返回值类型的自动推演

到了 C++14 之后，对于普通函数的返回值自动推演，可以通过 auto 来完成，比如：

```
auto f() { return Foo{1}.a; } // 返回值类型为int
```

当然，如果希望返回值类型运用 decltype 规则，则可以用 decltype(auto) 。比如：

```
auto f() -> decltype(auto) { // 返回值为int&&
  return (Foo{1}.a);
}
```

5.8 非类型模版参数

```
template <auto V>
struct C
{
   // ....
};

C<10>   a; // C<int>
C<'c'>  b; // C<char>
C<true> c; // C<bool>
```

5.9 函数模版的便捷写法

```
template <typename T1, typename T2>
auto add(T1 lhs, T2 rhs) {
   return lhs + rhs;
}
```

到了 C++20 ，允许让普通函数可以有更加便捷的写法：

```
auto add(auto lhs, auto rhs) {
   return lhs + rhs;
}
```

当然，如果你想指明两个参数属于同一种类型，但另外的参数没有这样的约束，则仍然需要写模版头：

```
template <typename T>
auto f(T a, auto b, T c, auto d); // a, c 必须同一类型，b, d 各自有各自类型
```

其等价于：

```
template <typename T, typename T1, typename T2>
auto f(T a, T1 b, T c, T2 d);
```

CHAPTER 6

初始化

C++ 的初始化方式之繁多，估计是所有编程语言之最。我们先通过一个列表来感受一下：

- 默认初始化
- 值初始化
- 直接初始化
- 拷贝初始化
- 零初始化
- 聚合初始化
- 引用初始化
- 常量初始化
- 数组初始化
- 列表初始化

以至于仅仅是初始化，都被专门开发成了一门课程。

但这些看似繁杂的初始化方式，背后有没有一些简单的线索可循？

6.1 直接初始化

我们先来看看最为简单的 **直接初始化**。

我们首先定义一个类 Foo：

```
struct Foo {
  enum class A {
    NIL,
    ANY,
    ALL
  };

  Foo(int a) : a{a}, b{true} {}
  Foo(int a, bool b) : a{a}, b{b} {}

  auto operator==(Foo const& rhs) const -> bool {
      return a == rhs.a && b == rhs.b;
  }

private:
  int a;
  bool b;
};
```

下面列表中所包含的构造表达式均为 **直接初始化**：

```
Foo object(1);

Foo object(2, false);

Foo object2(object);

Foo(1) == Foo(1, true);

new Foo(1, false);

long long a{10};
(long long){10} + a;

char b(10);
char(20) + b;

char* p{&b};
```

(下页继续)

(续上页)

```
Foo::A e{Foo::A::ANY};
```

简单说，当初始化参数非空时（至少有一个参数），如果你

1. 使用 **圆括号**初始化（构造）一个对象，或者
2. 用 **圆括号**或 **花括号**来初始化一个 *non-class* 类型的数据时（基本类型，指针，枚举等，因而只可能是单参）时，

这就是直接初始化。

这种初始化方式，对于 *non-class* 类型被称作 **直接初始化**很容易理解。而对于 *class* 类型，**直接初始化**的含义也很明确，就是直接匹配对应的构造函数。伴随着匹配的过程：

1. 参数允许窄向转换 (*narrowing*)；
2. 允许隐式转换；

比如:

```
long long a = 10;

Foo foo(a); // OK

struct Bar {
  Bar(int value) : value(value) {}
  operator int() { return value; }
private:
  int value;
};


Foo foo(Bar(10)); // Bar to int, OK
```

除此之外，还有几种表达式也属于 **直接初始化**：

1. *static_cast<T>(value)* ；
2. 使用 **圆括号**的类成员初始化列表；
3. *lambda* 的捕获初始化列表

6.2 列表初始化

不难看出，除了 *lambda* 的场景，以及用 **花括号**初始化 *non-class* 类型之外，**直接初始化**正是石器时代 (*C++ 11* 之前) 的经典初始化方式。

到了摩登时代 (自 *C++ 11* 起)，引入了被称作 *universal* 的统一初始化方式：列表初始化。之所以被称作 *universal* ，是因为之前花括号只被用来初始化聚合和数组，现在可以用来初始化一切: 基本类型，枚举，指针，引用，类。

由于列表为空有非常特殊而明确的定义, 我们在这里仅仅考虑列表非空的场景。

我们先看看如下表达式：

```
Foo foo{1, true};
Foo foo{2};

new Foo{3, false};

Foo{4} == Foo{4, true};
```

以及如下表达式：

```
Foo foo = {1, true};
Foo foo = {2};

Foo foo = Foo{3, false};
Foo foo = Foo{4};
```

这两组表达式都被称为 **列表初始化**。唯一的差别是，后者使用了等号，看起来像赋值一样。前者被称为 **列表直接初始化**，后者则叫做 **列表拷贝初始化**。

虽然后者名字里有 **拷贝**二字，并不代表其背后真的会进行拷贝操作。仅仅是因为历史的原因，以及为了给出两个名字以区分两种方式。

但事实上，对于 *class* 的场景，两者都是直接匹配并调用类的构造函数，并无根本差异。

其中一点细微的差别是：如果匹配到的构造函数，或者类型转换的 `operator T` 被声明为 `explicit` ，一旦你使用等号，则必须明确的进行指明：

```
struct Bar {
  explicit Bar(int a) {}
};


Bar bar = {10};     // fail
Bar bar = Bar{10}; // OK
Bar bar{10};        // OK
```

(下页继续)

(续上页)

```
struct Thing {
  explicit operator Bar() {  ...  }
};

Thing thing;

Bar bar = thing;       // fail
Bar bar = Bar{thing}; // OK
Bar bar{thing};        // OK
```

对于类来说，而列表初始化（使用 **花括号**），相对于直接初始化（使用 **圆括号**），其差异主要体现在两个方面：

1. 如果类存在一个单一参数是 `std::initializer_list<T>` ，或者第一个参数是 `std::initializer_list<T>` ，但后续参数都有默认值，使用 **花括号**构造，总是会优先匹配初始化列表版本的构造函数。
2. **花括号**不允许窄向转换。

6.3 值初始化

值初始化，简单来说，就是用户不给出任何参数，直接用 **圆括号**或者 **花括号**进行的初始化：

```
int a{};

Bar bar{};
Bar bar = Bar();
Bar bar = Bar{};
Bar bar = {};

Foo() + Bar();

new Bar();
new Bar{};
```

注意，这里面没有 `Bar bar()` 的初始化形式。由于这样的形式与函数声明无法区分，因而被明确定义为这是一个名为 `bar` ，返回值类型为 `Bar` 的函数声明。

而在石器时代，为了能够进行 **值初始化**，只能使用 `Bar bar = Bar();` 的形式。而这种形式在当时的语意为：等号右侧实例化了一个临时变量，通过拷贝构造构造了等号左侧的 `bar` ，但当时编译器基本上都会将这个不必要的拷贝给优化掉。到了 `C++ 17` ，这类表达式的拷贝语意被终结。更详细的细节请参照值与对象 。

值初始化的最大好处是，无论你是一个对象，还是一个基本类型或指针，你总是可以得到初始化（这也是为

何被称作值初始化）：

1. 如果一个类有 **自定义默认构造函数**，则其直接被调用；
2. 如果一个类没有 **自定义默认构造**，但有一个系统自动生成的默认构造函数 (或用户明确声明为 `default` 的默认构造函数)，则系统会先将其对象内存完全清零（包括 *padding* ），随后，如果这个类的任何非静态成员有 **非平凡默认构造**的话，在调用这些默认构造；
3. 对于基本类型和指针，直接清零。

6.4 默认初始化

相对于程序员会直接给出 `()` 或者 `{}` 的 **值初始化**，虽然都是无参数初始化，**默认初始化**什么括号也不给：

```
int a;

Foo foo;

new Foo;
```

如果一个类有非平凡的默认构造函数，则会直接调用。否则什么都不做，让那么没有非平凡构造的成员的内存状态留在它们被分配时内存 (无论是在堆中还是栈中) 的状态。比如：

```
struct Foo {
   int a{};
   int b;
};

Foo foo; // foo.a = 0, foo.b 为对象分配时内存的状态。
```

或许有人会倡导不要使用 **默认初始化**，而是统统使用 **值初始化**。这在很多情况下都是正确的，但却并非全无代价。对于可平凡构造的对象而言，值初始化会导致整个对象清零，如果对象较大，而随后的过程，你肯定会对对象的内容一一赋值（做真正的初始化），那么清零的过程其实是一种不必要的浪费。这对于关注性能的项目，可能是一个 *concern* 。

6.5 拷贝初始化

拷贝初始化非常简单：

```
int a = 10; // 拷贝初始化
int b = a; // 拷贝初始化

Foo foo{10};
```

(下页继续)

(续上页)

```
Foo foo1 = foo; // 拷贝初始化


auto f(int value) -> int {
   return value; // 拷贝初始化
}


f(a);   // 对参数进行拷贝初始化
f(10);  // 对参数进行拷贝初始化
```

请注意，拷贝初始化并不意味着必然发生拷贝，随着历史的车轮滚滚向前，曾经以为属于拷贝语义的表达式，如今早已面目全非。

6.6 零初始化

零初始化，并非 *C++* 的某种语法形式，而是伴随着其它语法形式的行为定义。比如：

```
static int a;
```

这样的数据定义，最终必然会被放入 *bss* 数据段，从而在程序加载时，被 *loader* 全部清零。

再比如：

```
int a{};

int a = {};
```

这事实上是 **值初始化**的范畴，只不过其结果是清零。

重要:

- 无参数初始化有两种形式：**值初始化**（带有 `()` 或 `{}` ）和 **默认初始化**（无 `()` 或 `{}`）。前者会保证进行初始化（调用默认构造，或清零，或混合）；后者只会调用默认构造（如果是平凡的，则什么都不做）。
- 有参数初始化，可以通过 `()` 或者 `{}` 的方式进行，两者的差异在于后者更优先匹配初始化列表，以及窄向转换的约束。
- 在不使用 `()` 或者 `{}` 的场景下，使用 = 进行的初始化，属于 **拷贝初始化**。如果被初始化对象是一个 *class* 类型，**copy 构造**或 **move 构造**会被调用；在使用 `()` 或 `{}` 的场景下，在 *C++ 17* 之后，除了 `explicit` 的约束之外，和 **直接初始化**没有任何语义上的差异。

CHAPTER 7

六大金刚

任何一个 C++ 类，总会面临六大特殊函数的问题：

1. *default* 构造
2. *copy* 构造
3. *move* 构造
4. *copy* 赋值
5. *move* 赋值
6. 析构

这六大金刚，有着不同的分类方法。

首先一种分类方法是：

1. 构造三杰: *default/copy/move* 构造;
2. 赋值二将: *copy/move* 赋值;
3. 析构

另一种分类方法是：

1. 默认构造
2. *copy* 家族: *copy* 构造/赋值
3. *move* 家族: *move* 构造/赋值
4. 析构

在谈论不同的特性时, 不同的分类方法各自有其自己的意义。

而这六大金刚之间的关系，在如下层面互相影响:

- 存在性
- 可操作性
- 平凡性

7.1 存在性

所谓 **存在性**，单纯指在一个类中，它的定义是否存在，无论是用户自己定义的还是系统默认生成的。

对于任何一个特殊函数，其声明/定义首先分为两大类别:

1. 用户显式声明/定义
    - 用户自定义
    - 显式声明/定义为 `default`
    - 显式声明为 `delete`
2. 编译器隐式声明/定义
    - 隐式声明/定义为 `default`
    - 隐式声明为 `delete`

在用户显式定义的情况下, 对于任何一个特殊函数:

1. 如果用户显式定义了它（包括 `=default`），它都明确地存在。
2. 如果用户显式删除了它 (通过 `=delete`)，它都明确地不再存在。

如果用户没有显式定义，编译器根据规则（正是我们后续章节要讨论的)，决定隐式的定义或删除它（二者必居其一)。

在我们后续讨论的规则之下，当编译器决定隐式定义某个特殊函数 (`=defalt`)，但此时，依然会面临无法生成的困境。比如，其某个非静态成员变量，或者某个父类将那个特殊函数删除了，或者访问被禁止了，则系统也会放弃对此特殊函数的生成，而隐式的将其声明为 `delete` 。这类情况属于共用规则，我们后面就不再专门进行讨论。

7.1.1 默认构造

只要用户 **显式声明**了构造函数列表（包括 **copy/move 构造**），系统就不会隐式定义 **默认构造**。

注意，用户 **显式声明**，并不是指用户 **自定义**: 用户可以明确地声明 `T() = default` 或者 `T() = delete`，但这些都不是用户自定义默认构造。但只要用户显式声明了 **任何**构造，编译器都不会再隐式生成默认构造。比如:

```
struct Thing {
   Thing(Thing&&) = delete;
};
```

在这个用户明确声明的构造函数列表中，并不能查到 **默认构造**，因而其并不存在。

如果用户没有明确声明 **任何**构造函数。编译器将会尽力为它生成一个。除非编译器发现完全无法做到。

> **注意:** 系统隐式定义的默认构造为 `T() = default` 。它从行为上与用户亲自定义一个空的默认构造函数: `T() {}` 没有任何差别: 调用父类和所有非静态成员的默认构造。所有的基本类型，指针，枚举等，其默认构造什么也不做，其值保持在其被分配出来时，内存中的样子。
>
> 不过，虽然 `T() = default` 和 `T() {}` 从自身行为上完全一致。但当用户对 `T` 类型的对象进行值初始化 时 (`T value{}`)，过程却完全不同: 前者，系统会对首先将 `value` 的内存清零，然后再调用 **默认构造**；而后者，由于用户提供了默认构造，系统则会直接调用默认构造。过程的不同，最终也会导致初始化的结果很可能不同。

7.1.2 copy 构造

copy 构造则在 **构造**三**杰**中，地位最高。

- 如果用户没有显式声明任何构造函数列表，系统会尽力为其生成一个。
- 如果用户显式声明了构造函数列表，即便其中查不到 **copy 构造**，但只要 **move 家族**的所有成员都没有被明确声明，编译器也会尽力生成一个 **copy 构造**。

```
struct Thing {
   Thing() {}
   // 隐式生成一个copy构造
   // Thing(Thing const&) = default;
};
```

```
struct Thing {
   Thing(Thing&&) = delete;
   // copy构造被删除
```

(下页继续)

(续上页)

```
    // Thing(Thing const&) = delete;
};
```

```
struct Thing {
    Thing(Thing&&) = default;
    // copy构造被删除
    // Thing(Thing const&) = delete;
};
```

```
struct Thing {
    auto operator=(Thing&&) -> Thing& = default;
    // copy构造被删除
    // Thing(Thing const&) = delete;
};
```

```
struct Thing {
    auto operator=(Thing&&) -> Thing& = delete;
    // copy构造被删除
    // Thing(Thing const&) = delete;
};
```

所以它的默认存在性，只受 **move 家族**的影响。

注解： 隐式生成的拷贝构造，会依次调用所有父类和非静态成员的 copy 构造。

7.1.3 move 构造

move 构造则在 **构造三杰**中，最为脆弱。

如果用户明确声明了如下任何一个，系统都不会自动生成 move 构造：

- *copy* 构造
- *copy* 赋值
- *move* 赋值
- 析构函数

所以其 **默认存在性**，不仅受 **copy 家族**和 **析构**的影响，还会遭受本家族另一成员的攻击。

copy 家族和 **move 家族**的这种互斥性，是因为它们从根本上属于同一范畴的问题 (参见右值引用)。一旦程序员打算对于这一范畴的问题做出自己的决定，那么编译器任何自作主张的行为都不能保证是安全的。因而，

move/copy 家族，编译器奉行的是 *nothing or all* 的策略：要么完全由编译器自动生成，要么完全由用户自己决定。

注解： 隐式生成的 move 构造，会依次调用所有父类和非静态成员的 move 构造。

7.1.4 copy 赋值

copy 赋值与 **copy 构造**的处境一致。

事实上，虽然 **copy 家族**的地位比 **move 家族**要高：**copy 家族**不受 **析构**的影响，也不会在本家族内自相残杀。但规范仍然倾向于让 **copy 家族**的地位降低到与 **move 家族**一样。也就是说，如果析构函数被程序员自定义，或者删除；或者 copy 家族内的另一成员由用户明确声明，那么编译器应该放弃对其提供默认实现。

C++ 标准对其的描述如下：

D.9: The implicit definition of a copy constructor as defaulted is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor. The implicit definition of a copy assignment operator as defaulted is deprecated if the class has a user-declared copy constructor or a user-declared destructor. It is possible that future versions of C++ will specify that these implicit definitions are deleted.

但由于规范仅仅将此定义为 **废弃**（*deprecated*），而不是一种强制规定，所以编译器的现行实现依然让 **copy 家族**保持了比 **move 家族**更高的地位（*CLANG* 通过 *-Wdeprecated*，*GCC* 通过 *-Wdeprecated-copy* 可以给出告警）。

而按照 *C++* 的保守传统，从 **废弃**到 **禁止**恐怕将是一个非常漫长的过程（甚至可能永不发生）。一个典型的例子是：对 *bool* 的 ++ 演算，在 *C++ 98* 里就被明确废弃了。但这么一个简单的，很少有人使用（误用）的特性，直到 *C++ 17* 才被彻底禁止。对于 **copy 构造/赋值**这种使用广泛，波及面极大的特性，我很怀疑其最终会被禁止。

所以，规范的这种倾向性，更多的是建议程序员遵从 **The Rule Of All or Nothing**：对于 **copy/move 家族 + 析构**，要么全靠编译器默认生成，要么一旦对一个类考虑了其中一个，就应该同时考虑其它四个。

注解： 隐式生成的 copy 赋值，会依次调用所有父类和非静态成员的 copy 赋值。

7.1.5 move 赋值

move 赋值与 **move 构造**的处境一致。差别只在于家族内自相残杀的对手。

```
struct Thing {
   Thing(Thing&&) = default;
   // move赋值被删除
   // auto operator=(Thing&&) -> Thing& = delete;
};
```

注解: 隐式生成的 move 赋值，会依次调用所有父类和非静态成员的 move 赋值。

7.1.6 析构

析构在 **六大金刚**中，处于食物链的顶端: 它只可能影响别人的存在性，而其它五位的存在性对其毫无影响。

一旦用户明确自定义了 **析构**，则 **move 家族**就丧失了被编译器隐式生成的权利。除非程序员显式声明，否则，**move 家族**的两个成员都被标记为删除。

事实上，这背后的逻辑非常简单：`move` 的典型应用场景为：将速亡值 的内容移动给另外一个对象之后，自身很快就会被销毁，因而 move 操作与析构行为是高度相关的。如果 **析构**是自定义的，那么 `move` 也应该由程序员自定义；编译器自作主张的默认生成是不负责任的。而如果析构函数被程序员明确声明为删除，`move` 却继续存在，这很明显违背了 `move` 本身的意义。

析构对于 **copy 家族**与 **默认构造**的存在性没有影响, 即便 **析构**被明确标记为删除。因为只创建不删除的对象，通过拷贝构造，或者通过拷贝复制进行修改，从语义和操作上并无问题。

但正如之前提到的，用户自定义 **析构**对于 **copy 家族**没有影响，这纯粹是历史原因所导致的，规范现阶段将其定义为 **废弃**。因为在很多场景下，如果程序员自定义了析构，如果编译器仍然自动生成 `copy` 家族的默认实现，会带来非预期的潜在风险。比如，一个对象持有一个指向另一个动态分配的对象的指针，程序员自定义的析构函数里，会释放掉指针所指向的内存。但程序员忘记自定义相关的 `copy` 构造，而编译器默认生成的浅拷贝实现，最终会导致内存的重复释放，最终会引发系统的崩溃。

注解: 系统自动生成的析构，会依次调用父类以及所有非静态成员的析构。

7.2 可操作性

而 **可操作性**，指的是，一个类的对象，是否可以执行某种操作。其与 **存在性**高度相关，但又不完全相同。

7.2.1 并不 move 的 move

首先，一个类，**move 构造**可以不存在，却是 **可 move 构造**的（即 `Foo foo2{std::move(foo1)}` 是合法的表达式）。

这背后的原因不难理解。因为 `std::move` 操作仅仅是将一个表达式无条件变为右值引用。只要有一个构造函数能够匹配右值引用，那么这个类就是 **可 move 构造**的。毫无疑问 `operator=(Foo const&)` 形式的拷贝构造可以匹配右值引用，因而即便没有右值引用的构造函数，它依然是 **可 move 构造**的。

```
struct Foo {
   auto operator=(Foo const&) -> Foo& = default;
};

static_assert(std::is_copy_constructible_v<Foo>);
static_assert(std::is_move_constructible_v<Foo>);
```

其次，一个类的拷贝构造可以是 `operator=(Foo&)` 的形式，但这样的拷贝构造，即无法接受 `Foo const&`，也无法接受 `Foo&&`，因而，如果这个类仅仅提供了这种形式的拷贝构造函数，那么它既不是 *copy constructible* 的，也不是 *move constructible* 的。

```
struct Foo {
   Foo() = default;
   auto operator=(Foo&) -> Foo& = default;
};

static_assert(!std::is_copy_constructible_v<Foo>);
static_assert(!std::is_move_constructible_v<Foo>);
```

但注意，这个 **copy 构造**函数，依然可以匹配 *non-const* 左值引用。因而依然可以进行 **copy 构造**操作。

```
Foo foo{};
Foo foo2{foo};
```

因而，

1. `std::is_copy_constructible_v<T>` 测试 `T(T const&)` 是否是合法的；而
2. `std::is_move_constructible_v<T>` 测试的则是 `T(T&&)` 表达式的合法性。

由于 **可 move 构造**的条件并不意味着 `T(std::move(t))` 必然匹配的是 **move 构造**，这就会在某些情况下，由于程序员的疏忽而导致非期望的行为。比如：

```
struct Foo {
   Foo(int a) : p{new int(a)} {}

   Foo(Foo const& rhs) : p{new int(*rhs.p)} {}
   auto operator=(Foo const& rhs) -> Foo& {
     delete p; p = new int{*rhs.p};
     return *this;
   }

   Foo(Foo&& rhs) : p{rhs.p} { rhs.p = nullptr; }
   auto operator=(Foo&& rhs) -> Foo& {
     delete p; p = rhs.p; rhs.p = nullptr;
     return *this;
```

(下页继续)

(续上页)

```
    }

    ~Foo() { delete p; }

private:
    int* p;
};


struct Bar : Foo {
  using Foo::Foo;

  ~Bar() { /* do something */ }
};
```

在这个例子中，子类 Bar 由于自定了 **析构**函数，按照之前在存在性 里所讨论的，编译器将不会自动为 Bar 生成 **move 家族**的任何函数，但却会自动为 Bar 生成 **copy 家族**的函数：

```
struct Bar : Foo {
  using Foo::Foo;

  // copy家族的默认存在性不受影响
  // Bar(Bar const&) = default;
  // auto operator(Bar const&) -> Bar& = default;

  // 由于~Bar()被明确定义，因而move家族不再存在
  // Bar(Bar&&) = delete;
  // auto operator(Bar&&) -> Bar& = delete;

  ~Bar() { /* do something */ }
};
```

在这样的情况下，如下代码将会十分完美的通过编译：

```
Bar bar{10};
Bar bar2{std::move(bar)};
```

但系统的行为却不是我们所期待的。(可以通过打开编译器告警，避免这样的悄无声息)

7.2.2 析构 = delete

另外一个特殊情况则是：如果一个类的 **析构**被标记为 `delete`，并不妨碍存在性规则。比如我们将上例中的 `Bar` 修改为：

```
struct Bar : Foo {
  Bar() : Foo{10} {}

  // copy家族的默认存在性不受影响
  // Bar(Bar const&) = default;
  // auto operator(Bar const&) -> Bar& = default;

  // 由于~Bar()被明确声明为delete，因而move家族也不再存在
  // Bar(Bar&&) = delete;
  // auto operator(Bar&&) -> Bar& = delete;

  ~Bar() = delete;
};
```

此时，我们依然可以合法地编写如下代码：

```
Bar* bar  = new Bar{};
Bar* bar2 = new Bar{*bar};
Bar* bar3 = new Bar{std::move(*bar2)};
*bar2     = *bar3;
*bar3     = std::move(*bar);
```

但此时，所有构造相关的可操作性检验统统失败。

```
static_assert(!std::is_default_constructible_v<Bar>);
static_assert(!std::is_copy_constructible_v<Bar>);
static_assert(!std::is_move_constructible_v<Bar>);
```

这是因为，虽然对于动态分配的对象而言，可以只创建，不销毁；但对于一个非动态非配的值对象而言，销毁是个必然会经历的过程，一旦无法销毁，也就意味着不能创建。

但 **赋值二将**的 **可操作性**检验依然是成功的：

```
static_assert(std::is_copy_assignable_v<Bar>);
static_assert(std::is_move_assignable_v<Bar>);
```

这是因为，即便你是动态创建出来的永不销毁的对象，相互之间依然可以进行赋值操作。

7.3 平凡性

平凡性当然首先是基于 **可操作性**的。你只有首先具备可操作性，才能谈论一个操作是不是平凡的。

而六大金刚一旦是平凡的，那么它们的行为也可以很平凡的分为两类：

1. 对于 **析构**和 **默认构造**，什么也不用做；
2. 对于 **copy/move 家族**的四大金刚，等同于 `::memcpy`；

虽然规范中，对于 **平凡 copy 构造**，明确的说明了 *padding* 并不需要拷贝，但也并不禁止，但编译器基本上都会基于性能和简单性的考量，直接 `::memcpy` 了事。

为了探究平凡性，我们先构造一个无比平凡的类：

```
struct Thing {
   Thing() = default;

   Thing(Thing const&) = default;
   auto operator=(Thing const&) -> Thing& = default;

   Thing(Thing&&) = default;
   auto operator=(Thing&&) -> Thing& = default;

   ~Thing() = default;
};
```

你无法再定义一个比它还要平凡的类，这六大 `default` 行为，其实完全不需要写。因而，毫无意外，它们应该都能通过平凡性测试：

```
static_assert(std::is_trivially_default_constructible_v<Thing>);

static_assert(std::is_trivially_copy_constructible_v<Thing>);
static_assert(std::is_trivially_copy_assignable_v<Thing>);

static_assert(std::is_trivially_move_constructible_v<Thing>);
static_assert(std::is_trivially_move_assignable_v<Thing>);

static_assert(std::is_trivially_destructible_v<Thing>);
```

而 **析构**函数，继续在 **平凡性**领域表现其王者气质。

一旦我们将其变为 **明确定义**的：

```
struct Thing {
   Thing() = default;

```

(下页继续)

(续上页)

```
    Thing(Thing const&) = default;
    auto operator=(Thing const&) -> Thing& = default;

    Thing(Thing&&) = default;
    auto operator=(Thing&&) -> Thing& = default;

    ~Thing() {} // 明确定义
};
```

则所有的构造，马上变为非平凡的：

```
static_assert(!std::is_trivially_default_constructible_v<Thing>);
static_assert(!std::is_trivially_copy_constructible_v<Thing>);
static_assert(!std::is_trivially_move_constructible_v<Thing>);
```

如果我们将 **析构**定义为 `delete` ，那么连 **可操作性**都没有了，就更不用说操作的 **平凡性**了。

也就是说，只有当 **析构**是平凡的，那么三大构造才可能是平凡的。

这样的决策并不是在所有的场景下都必然合理。但出于保守的动机，这又是一个合理的选择。比如，我们定义如下一个类：

```
struct Foo {
    int fd;
    ~Foo() { if(a != 0) ::close(fd); }
};
```

单纯从数据成员，以及其它五大金刚看，这个类也平凡无比。但那个无比平凡的整数成员，事实上是一个文件描述符。析构函数会负责将其关闭。

对于这个类，其用户必须保证其构造时，都进行零初始化：

```
Foo foo{};
```

但这个类，也可能通过某种框架被使用。比如 `vector<Foo>` 。当你调用 `vector.emplace()` 时，`emplace` 的实现可以根据平凡性进行优化：

```
if constexpr(!std::is_trivially_default_constructible_v<T>) {
    elem[n] = {};
}
```

我们知道 `{}` 这种值初始化方式，会保证对象一定会被初始化，最不济也会将内存清 0。但如果一个对象的默认拷贝函数是平凡的，我们则无需进行这样的重量级操作。直接用默认初始化——什么都不用做就好。

当然，对于非平凡默认构造的对象而言，还是要老老实实说进行值初始化为好。所以，对于 `Foo` ，系统必须明确的指明其默认构造是非平凡的，才可能让框架对其进行必要的初始化。

当然，你肯定会鄙视这个类的设计者，认为这是一个连菜鸟都不会做出的糟糕设计。但做为语言的设计者，却无法禁止程序员可以这么做。因而只能保守的决定，即便 **默认构造**、**拷贝构造**都是可操作的（甚至操作是平凡的），但如果你检测它是否是 **可平凡构造**的，它的答案是 NO。至少编译器或者框架基于 **平凡性** (而不是 **非平凡性**）所做出的任何自动决定都会被禁止。让程序员亲自为自己的设计决策负责。

另外，需要注意的是，**析构**的 **非平凡性**，并不会影响两个 **赋值**操作的 **平凡性**。对于上面的例子：

```
static_assert(std::is_trivially_copy_assignable_v<Thing>);
static_assert(std::is_trivially_move_assignable_v<Thing>);
```

> **注意:** 之所以两个赋值函数处处不受析构函数性质的影响（无论是存在性还是平凡性），核心原因在于：构造和析构是于对象的生命周期有关的接口，是必须存在的（尤其是构造），但两个赋值接口却是在对象存在的情况下的 **修改接口**（类似于 *set* 函数）。
>
> 一个只读对象可以没有 *set* 接口（也不应该有），但却不可能没有构造。它们和 **copy/move 构造**表面上的相似性, 经常会导致程序员忽略了它们从根本上不同的性质，从而本末倒置地陷入困惑。

除了析构函数之外，其它五大金刚的平凡性，则 **只受它们各自的影响**。如果它们各自本来是平凡的，将其中任何一个改为不平凡的（通过明确定义或 *delete* ），它自己就会变为非平凡的。但其它金刚的平凡性质保持不变。

除了这六大平凡性判断之外，还有两个总体判断平凡性的 *type trait* ：

1. std::is_trivially_copiable<T>
2. std::is_trivially<T>

其中前者包含了除了 **默认构造**之外的其它 ***五人帮***的平凡性判断：只有那五者都被判断为平凡的，才为真。

而后者，则必须 **六大金刚**统统是平凡的，才为真。

而前者对于框架尤其有价值的地方是：如果它断言为真，则使用 `::memcpy` 进行对象拷贝必然是安全的。但这并不意味着它断言为假，`::memcpy` 则是不安全的。毕竟那是一个在进一步信息缺失的情况下, 只能最苛刻保守地必然保证 *copy* 安全的条件。如果一个框架，能够获得更多的信息，则无需这么严苛的条件也可以进行安全的拷贝。而程序员自身是拥有信息最多的，上述五个条件即便一个都不成立，程序员也可能保证某个类 `::memcpy` 是安全的。

重要:

- 析构的平凡性影响所有构造的平凡性;
- 其它五者的平凡性各自独立;
- trivially_copiable 要求除了默认构造之外的其它五者必须平凡；trivial 则要求全部平凡。
- trivially_copiable 是在没有更多信息的前提下，也能保证拷贝安全。

CHAPTER 8

SFINAE

SFINAE 是 *Substitution Failure Is Not An Error* (**替换失败不是一个错误**) 的缩写。在 *C++11* 之前，这并不是一个正式的 *C++* 规范术语。而是由 *David Vandevoorde* 在其书籍 `C++ Templates: The Complete Guide` 首次命名的概念。

整个术语有三个关键词：

1. 替换
2. 失败
3. 错误

如果不能理解它们背后的准确含义，就很难正确的理解并应用 *SFINAE* 。

8.1 函数重载

之所以会存在 *SFINAE* ，关键在于两点：

1. *C++* 允许函数重载；
2. *C++* 有函数模版；

没有这两点，*SFINAE* 也无从谈起。

而允许 **函数重载**，就意味着，同一个函数名，可能存在多个版本的定义。

因而，当你的代码中出现 **函数调用**时，编译器需要弄明白，此 **函数调用**究竟调用的是那个版本。而这个弄明白的过程，被称作 *Overload Resolution* (**重载决议**) 。

其大致的过程如下：

1. 编译器首先会根据 *C++* 规范所定义的 **名字查找**规则，找到所有符合名字查找规则的同名函数，作为 **候选集**。如果 **候选集**为空，编译器直接报错。
2. 将明确不匹配的版本（比如参数个数不匹配）踢出 **候选集**。
3. 如果剩余的 **候选集**里存在 **函数模版**，则需要对 **模版参数**进行推演（如果调用时用户没有全部明确指定的话）。如果类型推演失败，则将此函数模版移出 **候选集**。如果类型推演成功，则将指定的，或推演出的类型，对模版参数进行 **替换**（*Substitution* ）。*SFINAE* 正是发生在这个环节：如果替换失败，编译器不会给出任何诊断信息，只是简单的将这个 **函数模版**踢出 **候选集**。如果替换成功，此模版函数就被实例化为一个普通函数。(**函数模版**自身并不是 **函数**)
4. 到这一步依然还剩下的 **候选集**，被称作 *viable candidates* （**可行候选集**）。编译器下一步的任务是从 **可行候选集**中找到 **最佳匹配**的版本。而这可能会导致三种结局：
   - 找到了 **最佳匹配**版本。编译器将选择这个版本。
   - **可行候选集**为空。这将导致编译错误，编译器会抱怨找不到合适的定义。
   - 存在超过一个 **最佳匹配**版本。这会导致 **二义性**，也会造成编译错误。
5. 如果找到了 **最佳匹配**版本，编译器还会进行其它检查（可见性，是否被声明为 `=delete` 等等）。

从这个过程我们就可以明确 *SFINAE* 的含义：某个函数调用的 **候选集**中，如果存在 **模版函数**，则会对模版参数进行推演并替换，而 **替换失败**不会直接导致 **编译错误**，只会导致编译静悄悄地将此 **模版函数**从 **候选集**中移出。

8.1.1 可行候选集

能够进入 **可行候选集**的候选函数必须满足如下特征：

1. 参数个数必须匹配。如果函数调用时程序员提供了 *M* 个参数，则候选函数参数个数必须属于如下三种情况之一：
   a. 有 *M* 个参数；
   b. 少于 *M* 个参数，但最后一个是变参；
   c. 多于 *M* 个参数，但第 *M+1* 及随后的参数都有默认值（或是变参）。
2. 对于每一个 **形参** (*parameter*)，调用时对应的 **实参**（*argument* ）都可以通过 **隐式转换** (*implicit conversion*) 转换为对应的 **形参**类型。如果 **形参**是引用类型，那么右值引用实参不能绑定到 *non-const* 左值引用形参；左值引用实参不能绑定到右值引用形参。

8.1.2 函数匹配度

为了选出 **最佳匹配**版本，**可行候选集**里任意两个函数必须能够对比 **函数匹配度**。

假设我们有两个函数 *A* 与 *B* ，

1. *A* 至少有一个参数比 *B* 更匹配，*B* 没有任何参数比 *A* 更匹配，则 *A* 比 *B* 更匹配；
2. 如果 *A* 与 *B* 的参数匹配度一样，*A* 不是函数模版，但 *B* 是，则 *A* 比 *B* 更匹配；
3. 如果 *A* 与 *B* 都是函数模版，但 *A* 比 *B* 更特化，则 *A* 比 *B* 更匹配。

8.1.3 参数匹配度

上述规则中的第一条，是通过对比每一个 **参数匹配度**，来决定 **函数匹配度**。而 **参数匹配度**的对比规则如下：

1. **标准转换 > 用户自定义转换** (通过 `operator` 定义的转换函数) **> 变参**。
2. 如果都是 **标准转换**，则
    a. 如果 *A* 的转换序列是 *B* 的转换序列的子序列，则 *A* 比 *B* 更匹配；否则，
    b. **精确匹配 > 数值向宽转换 > 类型转换**。

8.1.4 等价的函数模版

两个函数模版是等价的，如果：

1. 定义在相同的 *scope* 内；
2. 同名；
3. 模版参数等价，即：
    a. 模版参数个数一致；
    b. 每一对对应的模版参数的 *kind* 相同（都是类型，值，模版, 变参）；
    c. 如果某个参数是值，则类型应该相同；
    d. 如果某个参数是模版，模版应该等价；
4. 包含了模版参数的返回值类型和函数参数类型的表达式应该等价；

CHAPTER 9

C++ 编译时元编程（一）

C++ 编译时元编程，是对 **类型**和 **值**的计算。类型自不必说，不存在 mutable 的可能性；而数值，C++ 也规定了必须是常量，即 immutable 的数值即对象。而任何基于常量的计算，如果其计算能力是图灵完备的，那么其从实际效果上，必然与函数式编程是等价的。

为了理解 C++ 编译时元编程的思想，最好有一门函数式编程语言做为引导。最为知名的函数式编程语言 Haskell 是一个直觉上最佳的选择。不幸的是，由于 C++ 编译时，不仅仅可以操纵 **数值**，还可以操纵 **类型**，而这两者高度融合在一个体系中，没有清晰界限。但 Haskell 对于类型和数据区分的非常清楚：其每个数据都有自己的类型，但参与计算的是数据，没有计算类型的能力。而 Agda 的类型系统高了一个层次，在计算层面，数值和类型的边界被消除，拥有同等的计算能力。因而，本文主要通过 Agda 来做为引导，来理解 C++ 编译时元编程的思想。

9.1 List

我们先通过一段 Agda 代码来熟悉其语法和语意：

```
data List (A : Set) : Set where
  []    : List A
  _::_ : A -> List A -> List A
```

这段代码里，data 说明这是在定义一个数据类型，其名字是 List ，而这个类型，需要另外一个类型做为输入，即 A ，而 A 的类型为 Set ；而被定义的类型：List，其类型也是 Set 。

在类型系统理论中，每一个类型都是一个 **集合**，比如：bool 是一个集合，集合里的元素是其 **值域**：true 和 false 。而 int 则是整个整数集合。加法类型，即 C/C++ 中的 union ，其 **值域**则是 union 中每种类型

值域的并集（这就是为何被称做加法类型）；而乘法类型，即 C/C++ 中的 struct ，其 **值域**则是 struct 中每个 field 的 **值域**相乘。总之，不难理解，每一个类型都是一个集合。

但所有类型放在一起，也是一个集合。这个集合被称做 Set1 。之所以被叫做 Set1 ，因为理论上，所有的 Set1 也是一个集合，那个集合被称做 Set2 ，从而可以 Set3, ..., SetN, ... 无穷的递归上去。

当然，我们现实中的类型系统，到 Set1 即足够了。所以上面代码中的 Set ，就是 Set1 ，即所有类型的集合。因而，任何时候你看到一个符号的类型是 Set ，那就意味着那个符号是一个类型。

接着看上面的代码。第 2 行和第 3 行，分别定义了两个 *构造函数*：

1. 第一个构造了一个空列表，其类型当然是 List A ；
2. 第二个构造函数有两个参数：类型分别为 A 和 List A ，即把一个 A 类型的数据追加到列表前面，得到计算结果：一个新的列表。这就是 A -> List A -> List A 的含义。

如果我们把上述代码直接映射到 C++ ，就是下面的代码：

```
template <typename A>
struct List {
   List();
   List(A, List<A>);
};
```

请花上一点时间，仔细对照这两段代码。它们之间的语意映射关系相当直接和清楚。这其中，typename 正是 Agda 中的 Set 。另外，C++ 的构造函数的返回值类型正是其构造的类型本身，因而无须描述。

你或许会感叹 Agda 可以直接使用符号来定义构造。但那不是本文的重点。

当然，C++ 只是那么描述，是无法完成一个 List 真正的职责的。下面给出一个可以工作的实现：

```
template<typename T, size_t N = 0>
struct List {
  const T head;
  const List<T, N-1> tail;
};

template<typename T>
struct List<T, 0> {};
```

这里通过 *模版部分特化*的方式实现 if-else 逻辑。而这也正是函数式编程的主要方式：通过模式匹配选择不同的函数实现。比如：

```
fib   :: (Integral a) => a -> a
fib n = (fib n-1) + (fib n-2)
fib 1 = 1
fib 0 = 0
```

这是一段 Haskell 代码，其中 a 代表这是一种范型参数，其属于 Integral TypeClass 。而对应的 *C++* 实现为：

```
template <std::integral auto N>
            constexpr auto fib    = fib<N-1> + fib<N-2>;
template <> constexpr auto fib<1> = 1;
template <> constexpr auto fib<0> = 0;
```

其中 std::integral 是 C++20 引入的 concept ，它比 Haskell 的 TypeClass 更为强大。但在本例中，起到的作用一样。

但这个例子只是通过模式匹配在做 **数值**演算。而对于 List 的例子，我们则是通过模式匹配进行类型选择。从本质上理解，如果模糊数值和类型的差异，那么类模版也是一个函数。比如，下面的模版类：

```
template <typename T, int I>
struct Class {
   using type = T;
};
```

其语意上，是下面的类 agda 语法描述（并不是真正的语法，因为 agda 没有 struct/class ）：

```
Class     : (T : Set) -> int -> Set
Class T I = struct { using type = T; }
```

即，模版名字是函数名，其有两个参数，其中 T 通过花括号里 {T : Set} 说明 T 是一个类型。其参数为 T 和 int 类型的数值 I ，函数的求值结果是一个类型：即后面的结构体内容定义。

因而，上面的对于 List 的定义，转化为类 agda 语法为：

```
List     : (T : Set) -> size_t -> Set
List T N = struct { const T head; const List<T, N-1> tail; }
List T 0 = struct { }
```

通过不同的输入参数，我们匹配到了不同函数，返回了不同的类型。空列表为空实现，非空列表才会有 head 和 tail 。

将类模版理解为函数，将模版特化看作函数调用的模式匹配，这就把对于数值的计算，和对于类型的计算，完全统一在一起。这会极大的拓展对于类型操作能力的理解。要知道，C++ 范型对于类型的操作能力是 **图灵完备**的。不要只是把它当作简单的实例化一个容器那样的基本泛型，否则你会错过 C++ 最为强大，也最为精彩的能力。

我们接着实现上面的 List 。为了让用户可以不要在每次使用 List 时都要指明类型，我们定义两个 Deduction Guide ，而第 2 个正是 Agda 例子中的第 2 个构造。而另外一个构造，则是对只有一个元素情况下的简便写法。至于空列表构造，C++ 已经帮我们生成了默认构造，我们无须再写。

```
template<typename T>
List(T) -> List<T, 1>;
```

(下页继续)

(续上页)

```
template<typename T, size_t N>
List(T, List<T, N>) -> List<T, N+1>;
```

这里如果不使用 Deduction Guide，换成构造函数，则是如下写法：

```
template<typename T, size_t N = 0>
struct List {
  List(T head) : head{head}, tail{} {}
  List(T head, List<T, N-1> tail) : head{head}, tail{tail} {}

  const T head;
  const List<T, N-1> tail;
};
```

很明显，这种写法要🅵嗦的多。毕竟都是非常平凡的构造，写起来很无聊。另外，最重要的是，这样的写法，C++ 无法自动推演出来 N 的值。因为构造函数的参数 List<T, N-1> 里，N 处于一个计算表达式里。这在 C++ 的定义中属于不可推演上下文。

13.10.2.5 [temp.deduct.type]: A non-type template argument or an array bound in which a subexpression references a template parameter.

另外，Deduction Guide 本身并不是构造函数。此处之所以通过 Deduction Guide 就可以构造，是因为 List 类型本身是一个 **聚合** (Aggregate)，聚合本身就可以直接构造其成员。比如下面的聚合以及初始化：

```
struct Foo { int a; int b; };

Foo foo1 = { 1, 2 }; // a=1, b=2
Foo foo2 = { 1 };    // a=1, b=0
```

所以，Deduction Guide 并没有创建任何构造函数，而只是根据 Deduction Guide 的指导，在调用聚合的初始化而已。

下面我们再定义与空列表有关的类型和常量：

```
template<typename T>
using Nil_t = List<T, 0>;

template<typename T>
constexpr Nil_t<T> Nil{};
```

有了 Deduction Guide 的指导，和 Nil 常量的辅助，我们就可以定义 *List* 常量了：

```
constexpr auto emtpy = Nil<int>;                    // int␣
↪型空列表，由于类型无法推演，必须明确指明
```

(下页继续)

(续上页)

```
constexpr auto list1 = List{1, List{2, List{3}}};  // 构造 1::2::3::Nil
```

从中，你可以清晰的看出函数式语言中的 List 就是这样的递归构造。Agda 在构造一个 list 时，则是如下语法：

```
let emtpy = []
let list1 = 1 :: 2 :: 3 :: []
```

明显比我们上面的定义看起来要清晰。当然 C++ 也可以重载操作符，比如：

```
struct NilList {};
constexpr NilList nil = {};

template<typename A, size_t N>
constexpr auto operator>>=(const A& value, const List<A, N>& list) -> List<A, N+1> {
  return List{ value, list };
}

template<typename A>
constexpr auto operator>>=(const A& value, const NilList&) -> List<A, 1> {
   return List{ value };
}
```

然后，我们就可以做和 Agda 类似的写法：

```
constexpr auto list1 = 1 >>= 2 >>= 3 >>= nil;
```

之所以选择这个符号，因为 *C++* 只有 @= (其中 @ 代表 +, - , >> 等二元操作符）是右结合的。

另外，你会发现 nil 没有指明任何类型信息。而不像之前必须指明类型：Nil<int> 。这是因为，在它所在的 operator>>= 环境里，List 的类型可以从做左边的操作数 3 获取到。可以回到 operator>>= 里理解这一点。如果没有上下文可以推演类型，则仍然必须亲自指明类型。

如果你还想更加简洁，则可以使用变参模版大法：

```
template<auto H, auto ... RESTs>
constexpr auto makeList                    = List{H};

template<auto H, auto H1, auto ... RESTs>
constexpr auto makeList<H, H1, RESTs...>  = H >>= makeList<H1, RESTs...>;
```

这是一个完全递归的计算，典型的函数式计算方式。另外，makeList 从参数上约束了必须至少有一个元素，否则在空列表的情况下，其类型由于缺乏上下文而无法推导。

现在，用户就可以非常简单的创建列表了：

```
constexpr auto list1 = makeList<1,2,3,4>;
```

下面我们来看与 List 有关的操作。比如最典型的 map 操作。下面是 Agda 的实现：

```
map             : {A B : Set} -> (A -> B) -> List A -> List B
map f []        = []
map f (x :: xs) = f x :: map f xs
```

第一行类型声明。其意思是：有两个类型 A 和 B，函数的输入参数有两个：第一个参数 (A->B)，这是从 A 类型到 B 类型的映射函数，List A 是一个元素为 A 类型的 List，函数的求值结果是元素类型为 B 的 List。

C++ 的实现非常类似：

```
template<typename A, typename B>
constexpr auto map(Nil_t<A>, auto (*f) (A) -> B) -> Nil_t<B> {
   return Nil<B>;
}

template<typename A, typename B, size_t N>
constexpr auto map(List<A, N> xs, auto (*f) (A) -> B) -> List<B, N> {
   return f(xs.head) >>= map(xs.tail, f);
}
```

然后，你就可以这样使用：

```
constexpr auto result = map(makeList<1,2,3>, +[](int value) {
   return double(value + 1) * 1.2;
});
```

你或许已经注意到，我们定义的所有变量和函数都有 constexpr 声明，因为我们在做编译时的计算，只能是常量。编译时计算是完全没有副作用的。并且如果你的计算代码使用了任何标准中的未定义行为，都会导致编译出错。运行时计算则不会如此。

现在，我们再来定义另外一个 List 操作函数：将两个 List 衔接在一起。我们先来看 Agda 的实现：

```
_++_            : {A : Set} -> List A -> List A -> List A
[] ++ ys        = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

这个函数的类型很容易懂，不再赘述。其中新的元素是 _++_，这是这个函数的名字，两边的下划线说明这是一个中位操作符。所以其下面定义函数实现时，也直接使用了中位操作方式。

C++ 的实现则是重载操作符。但算法一摸一样：

```
template<typename A, size_t N>
constexpr auto operator+(const Nil_t<A>&, const List<A, N>& rhs) -> List<A, N> {
```

(下页继续)

(续上页)

```
    return rhs;
}

template<typename A, size_t M, size_t N>
constexpr auto operator+(const List<A, M>& lhs, const List<A, N>& rhs) -> List<A, M +␣
↪N> {
    return lhs.head >>= (lhs.tail + rhs);
}
```

注意，C++ 的实现里，模版参数多了 size 参数，因为它是 C++ 实现的 List 类型的一部分，但它属于自动推演参数，用户永远不需要亲自指定它。

然后用户就可以这么使用：

```
constexpr auto result1 = nil + makeList<1, 2, 3>;
constexpr auto result2 = makeList<'a', 'b', 'c'> + makeList<'d','e'>;
```

本文中 List 的例子，展示了在 C++ 编译时元编程时，和函数式编程完全一样的思路。当然，你永远不会在现实项目中使用 List 这样低效的结构。有很多支持常量计算的数据结构才是你真正应该选择的。

重要:

- C++ 编译时元编程都是常量语意；
- **类型**与 **值**，在 C++ 编译时元编程的世界里，从概念上没有本质区别。typename 是类型的 Set 。
- C++ 的类模版也是函数语意；其求值结果的类型是 Set ，即类型；
- **模式匹配**，**递归**，是函数式编程处理条件选择和循环问题的典型手段；同样也是 C++ 编译时计算的主要手段。

9.2 TypeList

自从 C++11 引入了变参模版，极大的增强了范型编程的能力。这就意味着，我们在前面的 **数值**列表，现在有了 **类型**列表。因而，在实际项目中，我们也需要对类型列表有着和类型列表一样的操作。

9.2.1 Elem

比如，我们想从一个 **类型列表**中取出第 N 个类型：

```
template<size_t N, typename ... Ts>
struct Elem;

template<size_t N, typename H, typename ... Ts>
struct Elem<N, H, Ts...> {
   using type = typename Elem<N-1, Ts...>::type;
};

template<typename H, typename ... Ts>
struct Elem<0, H, Ts...> {
   using type = H;
};
```

然后，我们再定义一个别名，让用户使用时可以有更简单的表达式：

```
template<size_t N, typename ... Ts>
using Elem_t = typename Elem<N, Ts...>::type;
```

如下的 Agda 伪代码，反映了上述同样的算法：

```
Elem           :: size_t -> [Set] -> Set
Elem N (H::Ts) = Elem (N-1) Ts
Elem 0 (H::Ts) = H
```

当然，代码中没有明确应对 N 值超过列表长度的情况，在 C++ 下，这回导致一个编译错误。而这正是我们想要的结果。

另外，你如果仔细观察，会发现 Elem 模版里只有一个元素，即 type 。这种情况下，其实 Elem 类模版是没有价值的；因为它一则是静态的，即没有人会用它所实例化得到的类型去创建对象；二则，它里面只有一个静态元素，并不起到包的作用（当一个类，或者模版里有多个元素时，并且这些元素都是静态的，那么类某种程度就像是一个包，或者名字空间一样）。

因而，如果我们可以直接这么写，就可以让代码更加简洁：

```
template<size_t N, typename ... Ts>
using Elem = typename Elem<N, Ts...>;

template<size_t N, typename H, typename ... Ts>
using Elem<N, H, Ts...> = typename Elem<N-1, Ts...>;

template<typename H, typename ... Ts>
using Elem<0, H, Ts...> = H;
```

但很不幸，using 一个类型，在 C++ 里属于 **别名** (Alias)，而别名模版只是别名而已，不支持特化。

所以，我们还是只能使用类模版的方式。但对比两者，我们我们可以看出，我们真正关心的是其中的 type 。而不是外面的类模版。所以后者才更真实直接的在表达我们本来的语意。之所以采取前者那种间接表达方式，是因为 C++ 的限制。

或许未来 C++ 标准委员会意识到这是一种广泛的需求，因为这是 C++ 泛型编程极为常见的需求，标准库里到处都是这种用法。

事实上，在 C++11 之前，你只有两种模版类型：类模版和函数模版。因而，对于处理结构性问题时，只有类模版一条途径，哪怕只是简单的值计算。我们还是举之前 斐波那契数列的例子，在 C++11 之前，

```
template <unsigned int N>
struct Fib {
   static const unsigned int value = Fib<N-1>::value + fib<N-2>::value;
}

template <> struct Fib<1> {
   static const unsigned int value = 1;
}

template <> struct Fib<0> {
   static const unsigned int value = 0;
}
```

同样的，模版 Fib 存在的唯一目的，是为了提供静态的 value ，（正如上面为了提供静态的 type ）。那时候，C++ 也不支持 constexpr ，所以也无法通过函数来进行这样的编译时值计算。因而只能以这样的方式实现。

随后，C++11 引入了 constexpr ，而 C++14 终于提供了 **变量模版**。于是程序员终于可以在意图只是通过模版定义单个常量时，可以直接使用更加简洁的方式，直接表达的自己意图。

值的枷锁已经被解除，单个类型的枷锁或许很快也会被打破。要知道，在 C++11 之前，甚至 **类型别名模版** 都不支持。

但无论如何，我们必须明确，Elem 的例子里，如果将 Elem 看作一个函数，从语意上，函数真正的求值结果是里面的 type ， 而不是外面的 struct/class 。

9.2.2 Drop

现在，我们再来实现另外一个经典函数：drop，即将列表中的前 N 个元素抛弃掉之后所得到的列表。

这从算法上和 Elem 极为相似，唯一的差别是，Elem 只要第 N (从 0 开始计数) 的元素，而 Drop 则是抛弃掉前 N 个元素（此时 N 不像 Elem 的 N 一样是索引，而是一个总数），得到一个列表。

结果是单个类型，还是一个类型列表，对于设计的影响完全不同。因为 C++ 类型却可以指代，因而，才可以通过：

```
template<typename H, typename ... Ts>
struct Elem<0, H, Ts...> {
    using type = H; // 这里通过type指代
};

template<size_t N, typename ... Ts>
using Elem_t = typename Elem<N, Ts...>::type; // 这里访问那个指代
```

但不幸的是，C++ 类型列表无法指代。你无法通过诸如：using type... = Ts... 的方式，来指代一个从模版参数传入的参数列表。因而，也就无法直接返回一个结果列表。

我们在之前已经看到，模版参数可以是 **类型**或者 **数值**。还有一种允许的模版参数类型就是 **模版**。之前我们一直强调，模版的语意是求类型的 **函数**；这就意味着，一个函数的参数也可以是函数，而能将函数当作参数，或者能返回函数的函数，被称作高阶函数。而允许高阶函数是函数做为一等公民的关键特征。与之对应，模版也是 C++ 泛型编程的一等公民。

高阶函数 (Higher Order Function): In mathematics and computer science, a higher-order function is a function that does at least one of the following:

- takes one or more functions as arguments (i.e. procedural parameters),
- returns a function as its result.

All other functions are first-order functions

所以，对于刚才的问题，我们可以传入一个函数（也就是模版），其参数是一个变参（即类型列表），这样我们就可以把计算出来的参数列表做为结果传递给那个模版，由那个模版根据调用者的需要，随意处理。下面是我们的实现：

```
template<
   size_t                           N,
   template<typename ...> typename RESULT,
   typename                      ... Ts>
struct Drop;

template<
   size_t                           N,
   template<typename ...> typename RESULT,
```

(下页继续)

(续上页)

```
   typename                          H,
   typename                      ... Ts>
struct Drop<N, RESULT, H, Ts...> {
   using type = typename Drop<N-1, RESULT, Ts...>::type;
};

template<
   template<typename ...> typename RESULT,
   typename                          H,
   typename                      ... Ts>
struct Drop<0, RESULT, H, Ts...> {
   using type = RESULT<H, Ts...>;
};

template<
   template<typename ...> typename RESULT>
struct Drop<0, RESULT> {
   using type = RESULT<>;
};
```

其中，RESULT 就是用户指定的回调模版，而将 TypeList 传递给它之后，它就被实例化为一个类型，从而就得到了指代能力。

用 Agda 伪代码描述如下：

```
Drop           :: size_t -> ([Set] -> Set) -> [Set] -> Set
Drop N RESULT (H::Ts) = Drop (N-1) RESULT Ts
Drop 0 RESULT (H::Ts) = RESULT H::Ts
Drop 0 RESULT []      = Result []
```

当然，其实最后两种情况，从道理上是可以合并的，但是从 C++ 模式匹配的角度，合并之后，C++ 无法判断两种情况那种更特别。只有将 H，Ts... 展开写，C++ 才能对比它和前面那个模版的特化度。

然后我们就可以定义 Drop_t 别名，以简化用户的表达式：

```
template<
  size_t                            N,
  template<typename ...> typename RESULT,
  typename                      ... Ts>
using Drop_t = typename Drop<N, RESULT, Ts...>::type;
```

我们之前已经谈到，Elem 和 Drop 的算法完全一直，无非就是所要的结果不同。因而，我们可以废弃掉之前 Elem 的实现，复用 Drop 的实现，而复用的方式，是通过传入自己特定的 RESULT 函数。

```
template<typename ... Ts>
struct Head;

template<typename H, typename ... Ts>
struct Head<H, Ts...> {
   using type = H;
};

template<
  size_t N,
  typename ... Ts>
using Elem_t = Drop_tt<N, Head, Ts...>;
```

我们传入的 RESULT 函数就是 Head，它拿到了一个 TypeList 之后，只取出第一个，即 H，而把其余的全部都丢弃掉。而这正是如果利用这种回调机制操作类型的一个示例。

除了直接的回调方式，还有另外一种方式：先让 Drop 自身计算结束，然后返回给一个 **高阶模版**。用户可以在那个时候，通过这个 **高阶模版**，以回调的方式获取结果：

```
template<
   size_t       N,
   typename ... Ts>
struct Drop {
   template<template<typename ...> typename RESULT>
   using output = RESULT<>;
};

template<
   typename     H,
   typename ... Ts>
struct Drop<0, H, Ts...> {
   template<template<typename ...> typename RESULT>
   using output = RESULT<H, Ts...>;
};

template<
   size_t       N,
   typename     H,
   typename ... Ts>
struct Drop<N, H, Ts...> {
   template<template<typename ...> typename RESULT>
   using output = typename Drop<N-1, Ts...>::template output<RESULT>;
};
```

其中，模版 output 即是计算返回的高阶模版。而回调的时机则推迟到：

```
template<
  size_t                          N,
  template<typename ...> typename RESULT,
  typename                     ... Ts>
using Drop_t = typename details::Drop<N, Ts...>::template output<RESULT>;
```

这种返回 **高阶模版**的做法，让模版计算和 lambda 一样，拥有了 **闭包**，即计算时自由访问环境的能力。我们先来看一个 C++ lambda 的例子：

```
auto l1 =
[](int a, int b) -> int {
   return [=](int c) -> int {
      return a + b + c;  // 访问 a, b
   }
}

auto l2     = l1(10, 20); // l2 is a lambda
auto result = l2(30);
```

对比一下模版的例子：

```
template<
   size_t       N,
   typename     H,
   typename ... Ts>
struct Drop<N, H, Ts...> {
   template<template<typename ...> typename RESULT>
   using output = // 访问外围模版的 Ts...
     typename Drop<N-1, Ts...>::template output<RESULT>;
};

using T1 = Drop<1, int, double, char>;  // 调用Drop得到 T1 类型
using T2 = T1::template output<Result>; // 调用T1类型的 output模版，得到我们的结果
```

9.2.3 Transform

现在我们再实现另外一个非常经典的函数 map，由于在 C++ 中，map 在标准库里代表一个 k/v 容器，而将 fp 领域里的 map 称做 transform，我们也继续遵守这个习俗。

Transform 操作是将一组类型，通过一个转化函数，转化成另外一组相同数量的其它类型。用 Agda 伪代码描述如下：

```
Transform  :: [Set] -> (Set -> Set) -> [Set]
```

注意，这里面有两个 TypeList ，一个是输入，一个是输出。而两者的内容经过 (Set -> Set) 转换后，很可能是完全不同的。

虽然之前的 Drop 也是输入一个 TypeList ，输出一个 TypeList ，但后者只是前者的一部分。因而事实上只需要在一个 TypeList 上操作即可。

对于 Transform 算法来说，它一边从输入列表中读取单个元素，将其转化后，不断追加新生成的输出列表上。另外，由于类型列表无法指代，所以，不可能使用诸如下面的算法：

```
{- 后面的 map f xs 有一个返回输出，但在 `C++`` 下无法通过这种方式直接输出一个列表 -}
map f (x :: xs) = f x :: map f xs
```

因而，我们必须将输出的列表随时保存在模版参数上，然后将最终结果想 Drop 算法一样，传递用户的回调。

同时，C++ 有另外一个约束，即类模版的变参列表只允许有一个，并且必须放在最后。(不知道未来 C++ 是否可以放开这样的限制，如果两个类型列表中，存在一个非类型参数，其实是可以区分的)。所以这个宝贵的变参列表位置一旦留给输出，那么输入列表该放在何处?

答案是，将其保存在另外一个模版里，其操作方式，应该和我们最初定义 List 非常类似：

```
template<typename ... Ts>
struct TypeList {};

template<typename H, typename ... Ts>
struct TypeList<H, Ts...> {
   using Head = H;
   using Tail = TypeList<Ts...>;
};
```

这个定义精准的反映了一个 List 的本质，这是一个递归结构。因而，你总是可以通过如下方式读取任何元素：

```
using list = TypeList<int, double, char>;

list::head                      // int
list::tail::head                // double
list::tail::tail::head          // char
list::tail::tail::tail::head    // 编译出错

TypeList<>::head                // 编译出错
```

所以，我之前其实撒谎了。我一直在宣称 C++ 的 TypeList 无法指代。其实指的是 Ts... 形式的列表无法指代。但一旦将其保存在刚刚定义的 TypeList 里，它就可以指代了。

但是，以 TypeList<Ts...> 的方式给用户，但需用需要的是 Ts... ，最终用户还是必须通过回调的方式才能使用 Ts... 形式，（这个我们后面会谈到），既然迟早都要回调，那还不如让用户完全意识不到 TypeList<Ts...> 这样一个中间结构，尽早回调。

下面就是 Transform 的实现：

```
template<
   typename                    IN,
   template<typename> typename F,
   typename = void,
   typename                ... OUT>
struct Transform {
   using type = TypeList<OUT...>;
};

template<
   typename                    IN,
   template<typename> typename F,
   typename                ... OUT>
struct Transform<IN, F, std::void_t<typename IN::Head>, OUT...> {
   using type =
      typename Transform<
         typename IN::Tail,
         F,
         void,
         __TYPE_LIST_APPEND(OUT..., typename F<typename IN::Head>::type)
      >::type;
};
```

代码看起来很长，但只是只是参数列表很长，真正的有逻辑的地方只有三处：

1. 第一个模版是整个 Transform 结束时的情况。所以将 OUT... ，即转换最终得到的输出列表传递给用户的回调模版。
2. std::void_t<typename IN::Head> 是一个 SFINAE 条件，即要求输入列表中还有 Head ，如果还存在，则说明 IN 列表还没有遍历结束；因而，
3. OUT..., typename F<typename IN::Head>::type 将通过 F 转换后的类型，追加到输出列表 OUT 后面。注意，其中的宏 TYPE_LIST_APPEND 什么都没做，只是为了表明代码意图。

上述算饭，用 agda 伪代码表现即为：

```
Transform              : [Set] -> (Set -> Set) -> (Set -> Set) -> [Set]
Transform []     F OUT = OUT
Transform (x:xs) F OUT = Transform xs F (xs ++ [F x])
```

当然这只是内部实现，给用户提供的真正接口是：

```
template<
  template<typename>     typename F,
  template<typename ...> typename RESULT,
```

(下页继续)

(续上页)

```
  typename                        ... IN>
using Transform_t =
  typename Transform<
    TypeList < IN...>,           // 将 IN... 保存到 TypeList
    F,
    void                         // 为了SFINAE条件判断
    __EMPTY_OUTPUT_TYPE_LIST___ // 输出列表最初为空
  >::type::template output<RESULT>;
```

所以，用户真正提供的参数只有三个，F 转化函数，IN 输入列表，以及用来回传最终结果回调模版 RESULT 。而宏 __EMPTY_OUTPUT_TYPE_LIST___ 背后什么都没有，正如一个 Ts... 形式的列表如果为空是，就什么都没有一样，这样在阅读代码时，很容易忽略这里还有一个空参数。而通过 __EMPTY_OUTPUT_TYPE_LIST___ 则可以起到提示的作用。

Transform 非常有用，比如，我们想给一个类型加上 Wrapper ，

```
template <typename T> struct Wrapper {...};

template <typename ... Ts>
struct Bar {
   // Ts... 里每一个类型都是 Wrapper<T> 的形式。
};

template <typename ... Ts>
struct Foo {
   template <typename T>
   struct AddWrapper { using type = Wrapper<T>; };

   using bar = Transform_t<AddWrapper, Bar, Ts...>;
   // ...
}
```

或者，识别出一组类型里，所有继承自某个类的类型：

```
template <typename T, typename = void>
struct ActionTrait {
   using type = void;
};

template <typename T>
struct ActionTrait<T, std::enabled_if_t<std::is_base_class_v<Action, T>>> {
   using type = T;
};
```

(下页继续)

(续上页)

```
template <typename ... Ts>
struct Bar {
   // Ts... 里，要么是 void，要么是Action的子类
};

template <typename ... Ts>
struct Foo {
   using bar = Transform_t<ActionTrait, Bar, Ts...>;
   // ...
}
```

9.2.4 Split

现在我们再来实现 split 。其语意是在第 N 个位置，将一个输入列表一分为二。

这个需求，又增加了新的困难，Transform 只要求输出一个列表，但这要要求输出两个。而模版的变参列表只允许有一个。怎么办?

首先，在有输出的情况下，用户必然是要传入回调。既然现在有两个输出，那么用户自然就需要两个回调。既然一个模版只允许有一个变参，那我们就分别两个回调传递个两个变参模版，一个给 TypeList<Ts...>，一个给计算模版。

所以，我需要先改造我们的 TypeList<Ts...> ，重新定义新的形式:

```
template<typename ... Ts>
struct TypeList {
  template<template <typename ...> typename RESULT>
  using output = RESULT<>;
};

template<typename H, typename ... Ts>
struct TypeList<H, Ts...> {
  using Head = H;
  using Tail = TypeList<Ts...>;

  template<template <typename ...> typename RESULT>
  using output = RESULT<H, Ts...>;
};
```

注意，这与之前的 TypeList 的唯一差别是多了一个高阶模版 output ，其用法和语意与我们在 Transform 一节讨论的一样。

而 Split 的实现，则关注在另外一个输出列表上:

```
template<
   size_t        N,
   typename      IN,
   typename ... OUT>
struct Split {
   using type = typename Split<
         N - 1,
         typename IN::Tail,
         __TYPE_LIST_APPEND(OUT..., typename IN::Head)
      >::type;
};

template<
   typename      IN,
   typename ... OUT>
struct Split<0, IN, OUT...> {
   struct type {
      using first  = TypeList<OUT...>;
      using second = IN;
   };
};
```

第一个模版处理的是 N 还没有递减到 0 的中间过程，所以继续递归。注意，那里有两个递归：一是 typename IN::Tail ，代表输入列表也在不断的通过递归抛弃 Head ；而另一个递归则是 Split 自身的递归，__TYPE_LIST_APPEND(OUT..., typename IN::Head) 则是将 IN 抛弃的 Head 拿过来，追加到第一个输出列表的后边。

所以，每一次递归，都像是游标在原始列表上移动，不断把后半部分的第一个元素，变为前半部分的最后一个元素。

而第二个模版则是已经到达了分割点，需要生成输出结果。因为有两个输出，因而分别被定义为 first 和 second 。前一个，是将生成的 OUT... 打包传递给 RESULT_1 ，后半部分，则向 IN 索要；而 IN 正是我们之前定义的 TypeList 。

最终，上述的内部算法，在如下代码处得到应用：

```
template<
   size_t N,
   template<typename ...> typename RESULT_1,
   template<typename ...> typename RESULT_2,
   typename ... IN>
class Splitter {
   using RawType =
     type_list::Split_t
         < N
```

(下页继续)

(续上页)

```
        , TypeList<IN...>                   // 将IN...传入，得到输入列表
        __EMPTY_OUTPUT_TYPE_LIST___         // 前半部分最初列表为空
        >;
public:
   using type = __TL_make_pair(typename RawType::first::template output<RESULT_1>,
                               typename RawType::second::template output<RESULT_2>);
};

template<
   size_t N,
   template<typename ...> typename RESULT_1,
   template<typename ...> typename RESULT_2,
   typename ... IN>
using Split_t = typename Splitter<N, RESULT_1, RESULT_2, IN...>::type;
```

Split_t 的输入参数，清晰的反映了用户需要指定的信息: N 分割的位置；RESULT_1, RESULT_2 分别为分割后两个部分的回调。IN 则是需要分割的输入列表。

重要:

- **类模版**的参数列表只允许有一个变参列表；因而，当需要多个变参列表时，则需要通过其它模版相助。

9.2.5 Fold

fold 是 list 非常重要的一个操作。其语意是遍历整个 list ，两两计算，最终得到一个结果。

它有两个版本：从左边 fold ，还是从右边 fold ，即:

```
(((x[1] op x[2]) op x[3]) op ...)       // 从左边开始fold
(... op (x[n-2] op (x[n-1] op x[n])))   // 从右边开始fold
```

而两个版本，都分为 **有初始值**和 **无初初始值**两种情况，即:

```
(((init op x[1]) op [x2]) op ...)
(... op (x[n-1] op (x[n] op init)))
```

C++17 提供了 fold expression ，从而让用户不再像过去一样，必须通过类模版的递归演算，来计算一个类型列表相关的值。比如:

```
template <typename ... Ts>
struct Foo {
   enum { Sum_Of_Size = (0 + ... + sizeof(Ts)) };
};
```

这里用到的是，有初识值 0 的从左边开始的，操作为 + 的 fold 计算。其结果是列表中所有类型的大小的总和。

从这个简单的例子可以看出，C++ 的 fold expression 不只在做 fold，还先做了 map，本例中即 sizeof。所以，其实际上是 map-reduce： 先 map ， 然后将 map 后的结果进行 fold 。

当然，对于 + 这种性质的计算，你也可以用从右边开始的 fold 。

```
template <typename ... Ts>
struct Foo {
   enum { Sum_Of_Size = (sizeof(Ts) + ... + 0) };
};
```

如果你可以确保，输入的类型列表非空，你甚至不用写初识值，因而更加简洁：

```
template <typename ... Ts>
struct Foo {
   enum { Sum_Of_Size = (sizeof(Ts) + ... ) };
};
```

但如果不能保证列表非空，则必须有初始值，否则，编译就会出错。(本身也的确让 fold 无法得到一个计算结果)

当然，fold 的能力，不会只是像上述的加法一样简单。比如，我想求所有类型尺寸的最大尺寸：

```
struct MaxSize {
   constexpr MaxSize(size_t size = 0) : size(size) {}
   constexpr operator size_t() const { return size; }
   size_t size = 0;
};

constexpr MaxSize operator<<(MaxSize maxSize, size_t size) {
   return std::max(maxSize.size, size);
}

template <typename ... Ts>
struct Foo {
   enum { Max_Size = ( MaxSize{} << ... << sizeof(Ts) ) };
};
```

这个例有三个关键点：

1. C++ 的 fold expression 的操作必须是 C++ 已经存在的二元操作符。几乎所有的二元操作符都可以用于 fold express 。如果一个二元操作符的 builtin 语意不满足你要求，或无法计算你的类型，你可以针对你的类型重载某个二元操作符。
2. 为了让重载的二元操作符不要影响其它类型，你必须定义一个自己的类型，在本例中，就是 MaxSize ；

3. 任何一个可以在被编译时计算使用到的函数必须是 constexpr 。本例子中，其构造函数，及类型转换函数，都必须是 constexpr 。当然，本例中，MaxSize 可以没有构造函数，那它将是一个 聚合。那么使用它的地方都必须是 聚合语法。比如：

```
struct MaxSize {
   constexpr operator size_t() const { return size; }
   size_t size = 0;
};

constexpr MaxSize operator<<(MaxSize maxSize, size_t size) {
   return MaxSize{ std::max(maxSize.size, size) };
}

template <typename ... Ts>
struct Foo {
   enum { Max_Size = ( MaxSize{} << ... << MaxSize{sizeof(Ts)} ) };
};
```

这使用起来，麻烦一些，所以我们为 MaxSize 提供了 constexpr 构造函数。

如果参与 fold 演算的类型到值的映射，不像直接调用 sizeof 这么简单，也可以使用模版变量来辅助：

```
template <typename ... Ts>
struct Foo {
   template <typename T>
   constexpr static size_t Size_Of  = std::is_base_of_v<Bar, T> ? sizeof(T) : 0;

   enum { Max_Size = ( MaxSize{} << ... << Size_Of<Ts> ) };
};
```

其算法是，只计算 Bar 子类的最大尺寸，其它类型则不参与计算。

运行时（而不是我们上述编译时例子）也可以使用 fold expression：

```
struct Node {
  int value;
  Node* left;
  Node* right;
  Node(int i=0) : value(i), left(nullptr), right(nullptr) {}
  ...
};

auto left = &Node::left;
auto right = &Node::right;
// traverse tree, using fold expression:
template<typename T, typename... TP>
```

(下页继续)

(续上页)

```
Node* traverse (T np, TP... paths) {
  return (np ->* ... ->* paths);
}
```

或者，我们想在输出对象的时候，在后面增加一个空格：

```
template<typename T>
class AddSpace {
   AddSpace(T const& r): ref(r) {}
   T const& ref;
};

auto operator<< (std::ostream& os, AddSpace<T> s) -> std::ostream& {
   return os << s.ref << ' '; // output passed argument and a space
}


template<typename... Args>
auto print (Args... args) {
   ( std::cout << ... << AddSpace(args) ) << '\n';
}
```

可以看出，C++17 推出的 `fold expression`，极大的简化了程序员对于 **变参**的计算。

但不幸的是，`fold expression` 只能计算 **数值**，而不能用来计算 **类型**（前面例子中与类型有关的计算，也是要把类型映射到数值）。

所以，我们必须得自己实现一个：

```
template
  < template<typename, typename> typename OP
  , typename                             ... Ts>
struct FoldR;

template
  < template<typename, typename> typename OP
  , typename                                 ACC
  , typename                             ... Ts>
struct FoldR<F, ACC, Ts...> {
   using type = ACC; // 列表为空，返回计算结果 ACC
};

template
  < template<typename, typename> typename OP
  , typename                                 ACC
```

(下页继续)

(续上页)

```
  , typename                                H
  , typename                            ... Ts>
struct FoldR<OP, ACC, H, Ts...> {
   using type = typename OP<ACC, typename FoldR<OP, H, Ts...>::type>::type;
};
```

这是一个右折叠算法。其中 OP 即二元操作，ACC 代表 **累计值**，Ts... 即输入列表。

其算法用伪代码描述，即为：

```
foldr                : {A B : Set} → (A → B → B) → B → List A → B
foldr op acc []      =  acc
foldr op acc (x::xs) =  x op (foldr op acc xs)
```

当然，从类型看，这个描述是计算值的，但无关紧要。

注意，FoldR 的主模版里是没有 ACC 的，这就意味着，主模版没有特别为初始值预留参数。但别担心，这样的实现，同时支持有初始值和无初始值两种场景：

```
template
  < template <typename, typename> typename OP
  , typename                              ... Ts>
using FoldR_t = typename details::FoldR<OP, Ts...>::type;

template
  < template <typename, typename> typename OP
  , typename                                INIT
  , typename                              ... Ts>
using FoldR_Init_t = FoldR_t<OP, Ts..., INIT>;
```

也就是说，当你有初始值时，做为 *右折叠*，只需要将初始值追加到类型列表最后即可。

下面是 *左折叠* 的实现：

```
template
  < template<typename, typename> typename OP
  , typename                              ... Ts>
struct FoldL;

template
  < template<typename, typename> typename OP
  , typename                                ACC
  , typename                              ... Ts>
struct FoldL<OP, ACC, Ts...> {
   using type = ACC;
```

(下页继续)

(续上页)

```
};

template
  < template<typename, typename> typename OP
  , typename                             ACC
  , typename                             H
  , typename                          ... Ts>
struct FoldL<OP, ACC, H, Ts...> {
   using type = typename FoldL<OP, typename OP<ACC, H>::type, Ts...>::type;
};
```

其算法用 Agda 代码描述如下：

```
foldr                : {A B : Set} -> (A -> B -> B) -> B -> List A -> B
foldr op acc []      =  acc
foldr op acc (x::xs) =  foldl op (op acc x) xs
```

注意，*左折叠*是一个 *尾递归* (tail recursion) 算法，因而可以进行优化，但 *右折叠*则不是，因为其必须层层递归，将右边的结果得到之后，才可以和左边一起进行 OP 计算，这从递归本质上，无法优化。

同样，*左折叠*主模版也没有 ACC ，因为只需要在有 INIT 的情况，将 INIT 放在列表头即可：

```
template
  < template <typename, typename> typename OP
  , typename                          ... Ts>
using FoldL_t = typename details::FoldL<OP, Ts...>::type;

template
  < template <typename, typename> typename OP
  , typename                             INIT
  , typename                          ... Ts>
using FoldL_Init_t = FoldL_t<OP, INIT, Ts...>;
```

有了 Fold ，我们就可以进行这样的计算：

```
template<typename T1, typename T2>
struct Combine {
   struct type : private T1, private T2 {
     auto createAction(ID id) -> Action* {
        auto action = T1::createAction(id);
        return action == nullptr ? T2::createAction(tid) : action;
   };
};

using type = FoldL_t<Combine, Ts...>;
```

从而将一些列的 ActionCreator 折叠成一个 ActionCreator 。

9.2.6 Flatten

比如，有这样一个结构：

```
Seq<int, Seq<double, Seq<short, float>, char>, long>
```

我们希望将内部的 Seq 展开到外部的 Seq 里，变为：

```
Seq<int, double, short, float, char, long>
```

这就是经典的 list 里有 list ，里面的 list 里可以再有 list ……，无论嵌套有多深，都可以通过 flatten 操作将其展开到最外层的 list 里。即将一个树状结构转化为一个平面结构。

稍加思考，我们就知道这是一个 fold 操作，即遍历整个列表，将每一个元素递归性的进行 flatten ，然后将得到到每个 list 不断连接，合成一个 list 。

我们知道 Fold 需要一个如下原型的 OP：

```
template <typename ACC, typename T>
struct OP;
```

即，将 T 和与前面的 ACC 进行某种计算，得到新的 ACC 做为结果。

而对于我们的 flatten 问题，传入的 T 需要 OP 将其展开成一个 Ts...（也可能只是 T ，如果其不是一个 List 的话）。

然后将得到的 T 或者 Ts... ，追加到 ACC 所代表的 Ts... 后面，得到一个新的 Ts... 。

这就意味着，ACC 自身要么是一个 Ts... ，要么 保存了一个 Ts... ，否则，上述算法不可能实现。

从 template <typename ACC, typename T> struct OP 的原型看，ACC 自身肯定不是 Ts... ，所以它只能 保存一个 Ts... ，并且这个 Ts.. 还可以和另外一个 Ts... 进行合并。

问题是，怎么可能让一个 类型 (原型里 ACC 是一个 typename ，代表 ACC 是 类型) 保存一个 Ts... ？我们之前早就讨论过，Ts... 是不可能直接指代的。

但我们也同样讨论过，模版具有 闭包的性质，我们可以利用这个性质，结合 高阶模版，就可以保存一个 Ts... ，并且这个 Ts... 可以和其它的 Ts... 进行合并。这就是下面的定义：

```
template<typename ... Ts1>
struct Accumulator {
   template<typename ... Ts2>
   using type = Accumulator<Ts1..., Ts2...>;
};
```

外面的是一个 Accumulator ，是一个 高阶模版，其参数用来 保存一个 Ts1... ，其返回的 type 是另外一个模版，其职责是将其环境中 Ts1... 和用户新传入的 Ts2... 进行连接，得到一个新的 Accumulator ，这个结构是不是很漂亮?

另外一个新的问题是，我们如何区分一个类型是可展开的？而另外一些类型不可以？很简单，要求可展开的类型都继承这样一个类：

```
struct FlattenableSignature {};

template<typename ... Ts>
struct Flattenable : FlattenableSignature {
   template<template<typename ...> typename RESULT>
   using OutputAllTypesTo = RESULT<Ts...>;
};
```

其中 FlattenableSignature 是一个签名类，用于判别，而 Flattenable 也是高阶模版，它让外界可以通过 OutputAllTypesTo ，以回调的方式得到它的 Ts... 。

另外，注意，无论是 FlattenableSignature ，还是高阶模版 Flattenable 都没有任何数据成员，也没有任何虚函数或虚基类，所以继承它们不会增加任何子类自身的开销。

下面定义我们真正符合 Fold 要求的 ACC：

```
template<typename Accumulator, typename T, typename = void>
struct FlattenAcc {
   using type = typename Accumulator::template type<T>;
};

template<typename Accumulator, typename T>
struct FlattenAcc
   < Accumulator
   , T
   , std::enable_if_t < std::is_base_of_v < CUB_NS::FlattenableSignature, T>>> {
   using type = typename T::template OutputAllTypesTo<Accumulator::template type>;
};
```

FlattenAcc 模版的前两个参数，正是 Fold 的 OP 原型所要求的两个参数，而第三个是在 C++20 之前（ C++20 直接使用 concept 即可），不得不使用的 SFINAE 技术，通过 std::is_base_of_v < CUB_NS::FlattenableSignature, T>> 来区分两种情况，而不得不额外引入的参数。

第一种情况，T 不是一个 Flattenable 的，那就直接将 T 合并到 Accumulator 里保存的 Ts... 里。

第二中情况，T 是一个 Flattenable 的，则将模版 Accumulator::template type 当作回调从 Flattenable 的高阶模版 OutputAllTypesTo 那里拿到其 Ts... ，而 Accumulator::template type 会将这两个 Ts... 进行衔接，并返回一个新的 Accumulator 。

而为了消除掉 FlattenAcc 那个额外的 void 参数：

```
template<typename ACC, typename T>
using FlattenAcc_t = FlattenAcc<ACC, T, void>;
```

这样我们就我们就可以直接调用 FoldL_Init_t：

```
using Acc = FoldL_Init_t<FlattenAcc_t, Accumulator<>, Ts...>;
```

其中，OP 是 FlattenAcc_t，由于初始时，结果列表为空，所以 INIT 是 Accumulator<>。而结算的结果，是一个 Accumulator<Ts...>，而为了取出其中的 Ts...，我们故伎重演，给它增加一个回调接口：

```
template<typename ... Ts>
struct Accumulator {
   template<typename ... NewTs>
   using type = Accumulator<Ts..., NewTs...>;

   // 取回 Ts... 的回调接口
   template<template<typename ...> typename RESULT>
   using output = RESULT<Ts...>;
};
```

然后，我们就可以将其输出到我们提供的回调模版 MyClass 上了：

```
template <typename ... Ts>
struct MyClass {
  // 回调之后，Ts ... 即 Flatten 之后的结果
  // 可以使用 Ts... 继续自己的计算
};

// result 是模版 MyClass ，被 Ts... 实例化后的类型
using result = typename Acc::template output<MyClass>;
```

9.3 pipeline

pipeline 事实上是函数的 compose 操作。其语意为：

```
(.) :: (b->c) -> (a->b) -> (a->c)
f . g = \x -> f (g x)
```

即 g 根据输入 x，计算出的结果，做为 f 的输入，并计算出最终结果。(在 Haskell 语法里，使用 () 表示这是一个中位符。)

而 pipeline 则是将一连串的函数，依此计算，前一个函数的计算结果，做为后一个函数的输入。对函数式编程而言，这是一种强大的基本抽象：一个个函数，通过 compose 组合在一起，完成更为复杂的计算。

但是，并非所有函数都是只有一个输入，所以，如果想让任何具有至少一个参数的函数都能被 compose，首先必须解决如何设置其它输入参数的问题（除了参与 pipeline 的那个参数）。一个设计良好的函数式编程语言，会支持 curry 。事实上，对于 haskell 这样的语言，其函数天然是 curry 的。对于一个 f :: a -> b -> c -> d 的函数，由于箭头是右结合的，所以其语意是 f :: a -> (b -> (c -> d))，即，你给出参数 a，求值结果是另外一个类型为 b -> (c -> d) 的函数；而你继续给出参数 b，则求值结果为类型为 c->d 的函数。

这就是给编程带来极大的便利，比如：

```
sum . takeWhile (<10000) . filter odd . map (^2) $ [1..]
```

首先产生一个无穷列表（$ 的优先级高于 .），然后这个无穷列表被传递给左边的整个 compose 之后的组合。

从右向左，map (^2)，filter odd，takeWhile (<10000)，每一个函数都给定了其它参数。最后，计算 sum 。

因而，如果想让 C++ 泛型计算支持 pipeline 首先要解决是 curry 的问题。

9.3.1 Curry

一个最简单的 curry 实现如下：

```
template <template <typename ...> F, typename ... ARGS>
struct Curry {
    template <typename ... MORE>
    using apply = F<ARGS..., MORE...>;
};
```

其背后的思想是：给定一个函数 F，同时给出其前面的一部分参数，然后返回另外一个函数 apply，其参数是 MORE 。

这样的实现，无法达到 haskell 每给出一个参数，估值结果还是一个函数的效果。因为这个实现只允许一次给出前面的参数，然后得到一个函数。如果你想继续给出一部分参数，就必须再次 Curry 。比如：

```
template <typename T1, typename T2, typename T3>
struct Foo {...};

Curry<Foo, int>::apply<double, char>; // Foo<int, double, char>

Curry<Curry<Foo, int>::apply, double>::apply<char>; // Foo<int, double, char>
```

注意，我们的 Curry 实现里，其输入参数 F，其参数列表是 typename ...，这样的原型可以匹配任意参数列表全部是 类型的模版。比如：

```
template <typename> struct S1;
template <typename, typename> struct S2;
```

(下页继续)

(续上页)

```
template <typename, typename ...> struct S1_N;
template <typename ...> struct S0_N;

Curry<S1>   // OK
Curry<S2>   // OK
Curry<S1_N> // OK
Curry<S0_N> // OK
```

但是，如果你的模版参数里包含有非类型参数，比如 int 或者 模版，则会导致编译失败。比如：

```
template <template <typename> typename> struct S1;
template <typename, int> struct S2;
template <auto, typename ...> struct S3;
template <int> struct S4;

Curry<S1>   // Fail
Curry<S2>   // Fail
Curry<S3>   // Fail
Curry<S4>   // Fail
```

接着回到我们的 Curry 。之前那个实现，不仅过于简单，甚至是不健壮的。比如，我们有如下定义：

```
template<typename T1, typename T2>
struct Foo {};

template<typename T1, typename T2>
using Foo_t = Foo<T1, T2>;
```

此时，如果我们对 Foo_t 使用我们的 Curry：

```
Curry<Foo_t, int>::apply<double> obj;
```

将会出现编译错误。

原因是，我们在使用别名模版时，对一个固定参数的 **别名模版**，在 Curry 里给其传入的变参。无论是 G++ 还是 clang 编译器都不允许这么做。除非将 **别名模版**也设计为变参。

这种检查发生在语意分析阶段。任何违背这种约束的代码都会被编译器查出。比如：

```
template<typename T1, typename T2>
using Foo_t = Foo<T1, T2>;

template <typename ... Ts>
using type = Foo<Ts...>;  // 错误。因为Foo_t是定长参数别名模版
```

(下页继续)

(续上页)

```
template <typename ... Ts>
struct Class {
   using type = Foo<Ts...>; // 错误。原因同上
};
```

不过，这样的限制仅仅是针对 **别名模版**，对于类模版，则完全没有这类限制。对于任何固定参数的类模版，你总是可以用一个变参包去实例化它。比如：

```
template<typename T1, typename T2> struct Foo {};

template <typename ... Ts>
using Foo_t = Foo<Ts...>; // Foo 定义为固定参数，但却可以给它传递一个变参包；
```

编译器会根据包实际展开之后，再检查是否符合类模版的需要。如果不匹配，则类型替换失败。

对于类模版，正是因为其对变参包实例化时才展开检查的性质。C++ 的变参模版灵活到了失去接口约束的作用，比如：

```
template <template <typename> F>
struct Service { ... };

template <typename ... Ts>
struct Client1 {...};

Service<Client1>  c;
```

这样的接口是满足正常的约束的。因为 Service 要求任何 Client 都提供能通过单参数回调的函数。但 Client 提供了一个变参，可以支持任意参数数量。这当然满足 Service 的要求。

但是，对于这种情况：

```
template <template <typename...> F>
struct Service { ... };

template <typename T>
struct Client1 {...};

Service<Client1>  c;
```

从设计语意上，Service 要求任何 Client 都必须提供一个可变参数的函数 F ，但 Client1 竟然只提供了单参数的版本。这很明显违背了契约。但 C++ 对于这种情况竟然也是允许的。

在 C++11 提案过程中，本来这种情况是不允许的。但后来发现提供这样的灵活度，可以给其它设计带来很大便利。比如：

```
template<template <typename ...> typename C, typename ... U>
struct Foo {
     using Container = C<U...>;
};

Foo<std::pair, int, double>::Container c; // std::pair<int, double>
```

所以，变参模版，可以匹配属于同一类别（比如都是值或者类型）的任意模版。而最终的正确性，则在实例化时再进行检查。

我们之前的 Curry 实现太简单，并且会因为固定参数的别名模版而导致错误。如果我们想追求可以多批次提供参数，而不想 curry 多次；并且要避免别名模版错误，则需要更加复杂的实现：

```
template <template <typename ...> typename F, typename ... ARGS>
struct Curry {
  template <typename ... MORE>
  using apply = Curry<F, ARGS..., MORE...>;
};

template <template <typename ...> typename F, typename ... ARGS>
requires requires { typename F<ARGS...>; }
struct Curry<F, ARGS...> {
   using type = F<ARGS...>;
};
```

这其中，使用了一个 concept ，来查看 F<ARGS...> 是否已经是一个类型（即其参数已经足够），如果是，则直接返回类型；否则，返回一个函数 apply ，当调用 apply 时，如果继续给出一部分参数，但对于 F 依然不够，则继续返回一个 Curry 。

然后，就可以这样使用：

```
Curry<Foo, int>::apply<double>::apply<char>::type; // Foo<int, double, char>
Curry<Foo, int, double>::apply<char>::type;        // Foo<int, double, char>
Curry<Foo, int>::apply<double, char>::type;        // Foo<int, double, char>
```

这个实现，也避免了之前实现的 **别名模版**问题。你一定会感到奇怪，对于这个版本的第二个特化：

```
template <template <typename ...> typename F, typename ... ARGS>
requires requires { typename F<ARGS...>; }
struct Curry<F, ARGS...> {
   using type = F<ARGS...>;
};
```

如果传入的 F 是一个定长参数 **别名模版**，里面不同样在给它传递变参吗?

这其中的差别在于，对于表达式 F<ARGS...> ，在模版实例化之前，编译器无从得知 F 是一个定长参数 **别**

名模版。因而，在模版的第一阶段检查中（未实例化之前的检查），无论从语法还是语意，这个表达式都没有任何问题。

随后，我们我们给出这样一个表达式：`Curry<Foo_t, int, double>`，在实例化的过程中，`F` 被替换为 `Foo_t`，而 `ARGS...` 则被替换为 `int, double`；此时，虽然编译器已经知道 `Foo_t` 是一个定长参数 **别名模版**，但用 `ARGS...` 此时已经不再是变参，而是非常具体的 `int, double`。因而，此时编译器只需要检查 `Foo_t<int, double>` 是否是一个合法的表达式即可。

而我们之前的版本：

```
template <template <typename ...> F, typename ... ARGS>
struct Curry {
    template <typename ... MORE>
    using apply = F<ARGS..., MORE...>;
};
```

在给出表达式 `Curry<Foo_t, int, double>` 之后，编译器同样进入替换阶段。将 `F` 替换为 `Foo_t`，将 `ARGS...` 替换为 `int, double`，然后我们就得到了表达式：

```
struct named_mangled_Curry {
    template <typename ... MORE>
    using apply = Foo_t<int, double, MORE...>;
};
```

对于 `apply` 模版，此时再次进行语意检查，发现定长参数 **别名模版** `Foo_t` 需要传递一个变参包 `MORE...`，这当然是一种语意错误。编译失败。

新版本的 `Curry` 避免了 **别名模版**错误，但却依然有一个未解决的问题。如果用户直接（`Curry` 时）或间接（`apply` 时）给出了超出 `Foo_t` 所需要的参数，则特化版本将永远也不可能被选择，只能选择主模版：

```
template <template <typename ...> typename F, typename ... ARGS>
struct Curry {
  template <typename ... MORE>
  using apply = Curry<F, ARGS..., MORE...>;
};
```

而这个模版，将永远也无法真正调用到 `F`，只能被困在虚幻的 `Curry` 世界里打转。

究其原因，是两种写法都没有解决关键的问题，传入的 `F` 究竟有几个参数?

在一个变参模版内部，计算一个变参包里的参数数量是非常简单的：`sizeof...(Ts)` 即可。但是对于传入的、可以匹配 `template <typename ...> typename F` 的模版，其参数个数可以是任意数目，比如：

```
template <typename T> struct C1;                          // 1个参数
template <typename T1, typename T2> struct C2;            // 2个参数
template <typename T1, typename ... Ts> struct C1s;       // 至少1个参数
```

(下页继续)

(续上页)

```
template <typename T1, typename T2, typename ... Ts> struct C2s; // 至少2个参数
template <typename ... Ts> struct Cs;                            // 任意多个参数
```

它们都可以匹配 F；并且没有任何直接的办法，站在 F 的角度求一个传入的模版的参数个数。

或许第一个跳到你脑海里的方法是模版萃取：

```
template<template<typename> typename>
struct TemplateTrait {
   constexpr static size_t num = 1;
};

template<>
struct TemplateTrait<template<typename, typename > typename> {
  constexpr static size_t num = 2;
}
```

或者：

```
template< template<typename> typename >
struct TemplateTrait {
   constexpr static size_t num = 1;
};

template < template< typename, typename > typename C>
struct TemplateTrait {
  constexpr static size_t num = 2;
}
```

很不幸，C++ 规定，模版的 **模版参数**不能用来做特化。而对于 **类模版**，不同的模版头，不允许存在两个主模版。

而高度灵活的函数模版这时候成了救世主。因为函数的重载非常灵活：两个同名函数可以除了名字一样，其它都不一样。比如，参数个数，参数类型。如果是模版的话，模版的参数列表也可以完全不同。所以，我们可以定义两个同名函数：

```
template <template<typename> typename>
auto DeduceArgs() -> Value<1>;

template <template<typename, typename> typename>
auto DeduceArgs() -> Value<2>;
```

然后，

```
template <typename T> struct C1;                    // 1个参数
template <typename T1, typename T2> struct C2;      // 2个参数

decltype(DeduceArgs<C1>())     // Value<1>
decltype(DeduceArgs<C2>())     // Value<2>
```

So far so good。但我们忘了一种重要的情况：变参。

```
template <typename T1, typename ... Ts> struct C1s;              // 至少1个参数
template <typename ... Ts> struct Cs;                            // 任意多个参数

decltype(DeduceArgs<C1s>())     // 二义性
decltype(DeduceArgs<Cs>())      // 二义性
```

正如我们之前所讨论的，这两个变参模版均可以匹配上面的两个 `DeduceTemplate` 函数。

你或许会想，是否可以补充一个变参版本，让它可以匹配变参场景，比如：

```
template <template<typename> typename>
auto DeduceArgs() -> Value<1>;

template <template<typename, typename> typename>
auto DeduceArgs() -> Value<2>;

template <template <typename...> typename>
auto DeduceArgs() -> Variadic; // 专门匹配变参
```

很不幸，这只会让情况变得更糟。因为变参版本可以匹配一切。有了它，所有的匹配都会出现二义性。

另外，需要特别强调的是，这几个函数都是主模版，不存在特化关系。并且，对于模版的 **模版参数**，是无法通过特化方式来进行区分的。所以，我们根本不需要尝试特化方案。

所以，现在核心的问题是：如何让函数模版能够在变参模版万能匹配的情况下，能够区分出定长参数和变参？

下面出场的就是 C++ 社区著名的惯用法：`tag dispatch` 。

我们先定义两个 `tag class`：

```
struct variadic_tag {};
struct fixed_tag : variadic_tag {};
```

你应该已经注意到，它们其中一个继承自另外一个。

然后，我们再定义两个重载函数，来使用这两个 `tag`：

```
template<template<typename...> typename F>
auto DeduceTemplateArgs(variadic_tag) -> Variadic;
```

(下页继续)

(续上页)

```
template<template<typename...> typename F>
auto DeduceTemplateArgs(fixed_tag)     -> decltype(DeduceArgs<F>());
```

现在，如果我们给出这样的表达式：`DeduceTemplateArgs<C1>(fixed_tag{})`，将会导致所有 **候选函数** 的的类型替换（将 `F` 替换为 `C1`），因而，第 2 个候选函数的返回类型变为 `decltype<DeduceArgs<C1>()>`；而其中的表达式 `DeduceArgs<C1>()` 则会进一步触发 `DeduceArgs` 的 **重载决议**（因为有多个候选）；而决议的结果找到了最佳匹配版本：

```
template <template<typename> typename>
auto DeduceArgs() -> Value<1>;
```

而其返回值类型为 `Value<1>`，所以 `decltype` 的结果也是 `Value<1>`。

做为最为匹配的版本，`DeduceTemplateArgs<C1>(fixed_tag) -> Value<1>` 是最终的选择。因而 `decltype(DeduceTemplateArgs<C1>(fixed_tag{}))` 的返回类型正是我们所需要的 `Value<1>`。

如果此时我们提供一个变参模版 `Cs`，那么表达式 ``DeduceTemplateArgs<Cs>(fixed_tag{}) 同样会触发上述过程；但 `DeduceArgs<Cs>()` 会因为二义性而决议失败，进而导致 `decltype(DeduceArgs<Cs>())` 失败，最终导致 `DeduceTemplateArgs(fixed_tag)` 从候选集中被排除出去。

此时，就只剩下 `DeduceTemplateArgs(variadic_tag)` 一个版本，但因为 `fixed_tag` 与 `variadic_tag` 之间的继承关系，所以这个版本也是匹配的，并且是现在候选集合中唯一的版本。所以，最终推演的结果是 `Variadic`，代表可变参数。

这里特别说明一下 `fixed_tag` 与 `variadic_tag`，我们之所以在表达式 `DeduceTemplateArgs<Cs>(fixed_tag{})` 里明确指明 `fixed_tag`，是因为我们希望优先使用第二个版本（在两个版本都匹配的情况下，`fixed_tag` 更为匹配）；只有在第二个版本失败的情况下，第一个版本才会得到选择。

在能够得知一个模版的参数个数的情况下，实现一个正确版本的 `Curry` 就变为可能。无论你选择上面的那种 `Curry` 实现，都可以结合参数个数信息，给出一个健壮的实现。在次就不再赘述。

9.4 Compose

`Compose` 的实现就简单很多。如下：

```
template <CallableConcept ... OPs>
struct Compose;

template <CallableConcept LAST>
struct Compose<LAST> : LAST {};
```

(下页继续)

(续上页)

```
template <CallableConcept H, CallableConcept ... OPs>
struct Compose<H, OPs...> {
   template <typename INPUT>
   using apply = typename Compose<OPs...>::template apply<typename H::template apply
↪<INPUT>>;
};
```

CallableConcept 要求每一个参与组合的 OP 都必须提供一个 apply 函数。之所以产生这样的约束，因为一则模版参数不能以变参的方式表达传递多个模版，而只能传递类型或值的变参包。二则，一个 Curry 的结果类型自身就是一个带有 apply 函数的类型。

而 compose 内部的算法，则非常直接，首先让 H 进行计算 typename H::template apply<INPUT>，其输出做为 Compose<OPs...>::template apply 的输入。

与 haskell 版本不同的是，其 compose 是从右向左，这符合数学函数表达习惯。而我们的实现是从左向右，这符合程序员对于 pipeline 的认知。

然后，我们将我们对 list 的各种操作进行 Curry，并用宏让一切看起来更简洁：

```
#define take(n)       Curry<Take, Value<n>>
#define drop(n)       Curry<Drop, Value<n>>
#define filter(f)     Curry<Filter, Value<f>>
#define sort(f)       Curry<Sort, Value<f>>
#define transform(f)  Curry<Transform, Value<f>>
```

之所以，有 Value 就是因为 Curry 要求一个模版的参数必须都是类型（因为 C++ 要求一个变参包里必须是同一种 kind）。

然后我们就可以在 pipeline 中直接使用它们：

```
Pipeline<List<1,2,3,4,5,6,7>,
   take(5),
   filter([](int n){ return n % 2 > 0; }),
   transform([](int n) { return n + 10; })>;
```

9.5 延迟估值

C++ 的泛型天然就是延迟估值的，因而你可以构造一个无穷列表：

```
template <typename T, T N, T STEP = 1>
struct InfiniteValueList {
   constexpr static T Head = N;
   using Tail = InfiniteValueList<T, N + STEP, STEP>;
};
```

(下页继续)

(续上页)

```
template <auto V>
struct RepeatValueList {
   constexpr static auto Head = V;
   using Tail = RepeatValueList<V>;
};
```

你不用担心这样的结构定义会无穷递归下去而导致编译器崩溃。并且你可以安全的写如下算法的代码：

```
Pipeline
  < InfiniteValueList<int, 1, 2>
  , Drop<2>
  , Take<5>>
  // 结果是 5, 7, 9, 11, 13

Pipeline
  < TypeList<int, double, char>
  , ZipWith<RepeatValueList<5>>>
  // 结果是 [(int, 5), (double, 5), (char, 5)]
```

9.6 Optional

Optional 是一个存在非常广泛的语意。比如，指针空与非空，非法值与合法值，存在与不存在…等等；在 Haskell 语言里，这种概念被称做 Maybe 。

而在类型的世界里，则可以将 void 看作 None （或 Nothing ），void 和其它类型一样，本身也是一个类型，但其值域为空。也就是说，你无法用它实例化任何数据。

所以，在对类型进行，transform，filter，或者 fold 操作时，你总是可以用 void 当作 None 语意。比如：

```
template <typename T, typename = void>
struct ActionTrait {
   using type = void;
};

template <typename T>
struct ActionTrait<T, std::enabled_if_t<std::is_base_class_v<Action, T>>> {
   using type = T;
};
```

其语意是，一个类型如果是 Action 的子类，返回的则是 Maybe<T> ，否则返回 Nothing 。

```
template <typename ... Ts>
struct Bar {
   // Ts... 里，全是Action的子类
};

typename Pipeline
         < TypeList<Ts...>
          , transform(ActionTrait)
          , filter(Maybe)
          >::template exportTo<Bar>;
```

甚至，你可以将对于 void 的过滤操作自动内嵌到操作中，比如，Transform 可以提供一个版本，自动抛弃掉结果为 void 的类型；或者，在 Fold 时，自动跳过是 void 的类型。但这都是优化的事情，即便不提供，由于有了 Filter ，也都可以完成计算。

重要:

- C++ 泛型计算，是完全函数式的；
- C++ 泛型，即可以计算数值，也可以计算类型，而两者都是 图灵完备的；
- C++ 泛型计算是 lazy 的；
- 模式匹配用来进行路径选择（辅助以 SFINAE 和 CONCEPT ），递归用来解决 循环问题；
- 模版是泛型计算的一等公民：对于 高阶模版的支持，及 闭包性质，可以将其理解为泛型计算时的 lambda 。
