12 | 编译期多态：泛型编程和模板入门

2019-12-23 吴咏炜

现代C++实战30讲

进入课程 >

讲述：吴咏炜
时长 13:19　大小 9.16M

你好，我是吴咏炜。

相信你对多态这个面向对象的特性应该是很熟悉了。我们今天来讲一个非常 C++ 的话题，编译期多态及其相关的 C++ 概念。

面向对象和多态

在面向对象的开发里，最基本的一个特性就是“多态” [1]——用相同的代码得到不同结果。以我们在 [第 1 讲] 提到过的 `shape` 类为例，它可能会定义一些通用的功能，然后在子类里进行实现或覆盖：

```
class shape {
public:
  …
  void draw(const position&) = 0;
};
```

复制代码

上面的类定义意味着所有的子类必须实现 `draw` 函数，所以可以认为 `shape` 是定义了一个接口（按 Java 的概念）。在面向对象的设计里，接口抽象了一些基本的行为，实现类里则去具体实现这些功能。当我们有着接口类的指针或引用时，我们实际可以唤起具体的实现类里的逻辑。比如，在一个绘图程序里，我们可以在用户选择一种形状时，把形状赋给一个 `shape` 的（智能）指针，在用户点击绘图区域时，执行 `draw` 操作。根据指针指向的形状不同，实际绘制出的可能是圆，可能是三角形，也可能是其他形状。

但这种面向对象的方式，并不是唯一一种实现多态的方式。在很多动态类型语言里，有所谓的“鸭子”类型 [2]：

> 如果一只鸟走起来像鸭子、游起泳来像鸭子、叫起来也像鸭子，那么这只鸟就可以被当作鸭子。

在这样的语言里，你可以不需要继承来实现 `circle`、`triangle` 等类，然后可以直接在这个类型的变量上调用 `draw` 方法。如果这个类型的对象没有 `draw` 方法，你就会在执行到 `draw()` 语句的时候得到一个错误（或异常）。

鸭子类型使得开发者可以不使用继承体系来灵活地实现一些“约定”，尤其是使得混合不同来源、使用不同对象继承体系的代码成为可能。唯一的要求只是，这些不同的对象有“共通”的成员函数。这些成员函数应当有相同的名字和相同结构的参数（并不要求参数类型相同）。

听起来很抽象？我们来看一下 C++ 中的具体例子。

容器类的共性

容器类是有很多共性的。其中，一个最最普遍的共性就是，容器类都有 `begin` 和 `end` 成员函数——这使得通用地遍历一个容器成为可能。容器类不必继承一个共同的 Container 基类，而我们仍然可以写出通用的遍历容器的代码，如使用基于范围的循环。

大部分容器是有 `size` 成员函数的，在“泛型”编程中，我们同样可以取得一个容器的大小，而不要求容器继承一个叫 SizeableContainer 的基类。

很多容器具有 `push_back` 成员函数，可以在尾部插入数据。同样，我们不需要一个叫 BackPushableContainer 的基类。在这个例子里，`push_back` 函数的参数显然是都不一样的，但明显，所有的 `push_back` 函数都只接收一个参数。

我们可以清晰看到的是，虽然 C++ 的标准容器没有对象继承关系，但彼此之间有着很多的同构性。这些同构性很难用继承体系来表达，也完全不必要用继承来表达。C++ 的模板，已经足够表达这些鸭子类型。

当然，作为一种静态类型语言，C++ 是不会在运行时才报告“没找到 `draw` 方法”这类问题的。这类错误可以在编译时直接捕获，更精确地来说，是在模板实例化的过程中。

下面我们通过几个例子，来完整地看一下模板的定义、实例化和特化。

C++ 模板

定义模板

学过算法的同学应该都知道求最大公约数的辗转相除法，代码大致如下：

复制代码

```
int my_gcd(int a, int b)
{
  while (b != 0) {
    int r = a % b;
    a = b;
    b = r;
  }
  return a;
}
```

这里只有一个小小的问题，C++ 的整数类型可不止 `int` 一种啊。为了让这个算法对像长整型这样的类型也生效，我们需要把它定义成一个模板：

```
template <typename E>
E my_gcd(E a, E b)
{
  while (b != E(0)) {
    E r = a % b;
    a = b;
    b = r;
  }
  return a;
}
```

这个代码里，基本上就是把 `int` 替换成了模板参数 `E`，并在函数的开头添加了模板的声明。我们对于“整数”这只鸭子的要求实际上是：

- 可以通过常量 `0` 来构造
- 可以拷贝（构造和赋值）
- 可以作不等于的比较
- 可以进行取余数的操作

对于标准的 `int`、`long`、`long long` 等类型及其对应的无符号类型，以上代码都能正常工作，并能得到正确的结果。

至于类模板的例子，我们可以直接参考 [第 2 讲] 中的智能指针，这儿就不再重复了。

实例化模板

不管是类模板还是函数模板，编译器在看到其定义时只能做最基本的语法检查，真正的类型检查要在实例化（instantiation）的时候才能做。一般而言，这也是编译器会报错的时候。

对于我们上面 `my_gcd` 的情况，如果提供的是一般的整数类型，那是不会有问题的。但如果我们提供一些其他类型的时候，就有可能出问题了。以 CLN，一个高精度数字库为例（注：我并不是推荐大家使用这个库），如果我们使用它的 `cl_I` 高精度整数类型来调用 `my_gcd` 的话，出错信息大致如下：

```
test.cpp: In instantiation of 'E my_gcd(E, E) [with E = cln::cl_I]':
test.cpp:20:29:   required from here
test.cpp:9:17: error: no match for 'operator%' (operand types are 'cln::
cl_I' and 'cln::cl_I')
         E r = a % b;
               ~~^~~
compilation terminated due to -Wfatal-errors.
```

其原因是，虽然它的整数类 `cl_I` 设计得很像普通的整数，但这个类的对象不支持 `%` 运算符。出错的第 20 行是我们调用 `my_gcd` 的位置，而第 9 行是函数模板定义中执行取余数操作的位置。

实例化失败的话，编译当然就出错退出了。如果成功的话，模板的实例就产生了。在整个的编译过程中，可能产生多个这样的（相同）实例，但最后链接时，会只剩下一个实例。这也是为什么 C++ 会有一个单一定义的规则：如果不同的编译单元看到不同的定义的话，那链接时使用哪个定义是不确定的，结果就可能会让人吃惊。

模板还可以显式实例化和外部实例化。如果我们在调用 `my_gcd` 之前进行显式实例化——即，使用 `template` 关键字并给出完整的类型来声明函数：

复制代码

```
1 template cln::cl_I
2   my_gcd(cln::cl_I, cln::cl_I);
```

那出错信息中的第二行就会显示要求实例化的位置。如果在显式实例化的形式之前加上 `extern` 的话，编译器就会认为这个模板已经在其他某个地方实例化，从而不再产生其定义（但代码用到的内联函数仍可能会导致实例化的发生，这个会随编译器和优化选项不同而变化）。在我们这个例子里，就意味着不会产生上面的编译错误信息了。当然，我们仍然会在链接时得到错误，因为我们并没有真正实例化这个模板。

类似的，当我们在使用 `vector<int>` 这样的表达式时，我们就在隐式地实例化 `vector<int>`。我们同样也可以选择用 `template class vector<int>;` 来显式实例化，或使用 `extern template class vector<int>;` 来告诉编译器不需要实例化。显式实例化和外部实例化通常在大型项目中可以用来集中模板的实例化，从而加速编译过程——不需要在每个用到模板的地方都进行实例化了——但这种方式有额外的管理开销，如

果实例化了不必要实例化的模板的话，反而会导致可执行文件变大。因而，显式实例化和外部实例化应当谨慎使用。

特化模板

如果遇到像前面 CLN 那样的情况，我们需要使用的模板参数类型，不能完全满足模板的要求，应该怎么办？

我们实际上有好几个选择：

添加代码，让那个类型支持所需要的操作（对成员函数无效）。

对于函数模板，可以直接针对那个类型进行重载。

对于类模板和函数模板，可以针对那个类型进行特化。

对于 `cln::cl_I` 不支持 `%` 运算符这种情况，恰好上面的三种方法我们都可以用。

一、添加 `operator%` 的实现：

复制代码

```
cln::cl_I
operator%(const cln::cl_I& lhs,
          const cln::cl_I& rhs)
{
  return mod(lhs, rhs);
}
```

在这个例子，这可能是最简单的解决方案了。但在很多情况下，尤其是对对象的成员函数有要求的情况下，这个方法不可行。

二、针对 `cl_I` 进行重载：

为通用起见，我不直接使用 `cl_I` 的 `mod` 函数，而用 `my_mod` 把 `my_gcd` 改造如下：

复制代码

```
template <typename E>
E my_gcd(E a, E b)
```

```
{
  while (b != E(0)) {
    E r = my_mod(a, b);
    a = b;
    b = r;
  }
  return a;
}
```

然后，一般情况的 `my_mod` 显然就是：

复制代码

```
template <typename E>
E my_mod(const E& lhs,
         const E& rhs)
{
  return lhs % rhs;
}
```

最后，针对 `cl_I` 类，我们可以重载（overload）：

复制代码

```
cln::cl_I
my_mod(const cln::cl_I& lhs,
       const cln::cl_I& rhs)
{
  return mod(lhs, rhs);
}
```

三、针对 `cl_I` 进行特化：

同二类似，但我们提供的不是一个重载，而是特化（specialization）：

复制代码

```
template <>
cln::cl_I my_mod<cln::cl_I>(
  const cln::cl_I& lhs,
  const cln::cl_I& rhs)
{
  return mod(lhs, rhs);
```

```
7 }
```

这个例子比较简单，特化和重载在行为上没有本质的区别。就一般而言，特化是一种更通用的技巧，最主要的原因是特化可以用在类模板和函数模板上，而重载只能用于函数。

不过，我只是展示了一种可能性而已。通用而言，Herb Sutter 给出了明确的建议：对函数使用重载，对类模板进行特化 [3]。

展示特化的更好的例子是 C++11 之前的静态断言。使用特化技巧可以大致实现 `static_assert` 的功能：

复制代码

```
template <bool>
struct compile_time_error;
template <>
struct compile_time_error<true> {};

#define STATIC_ASSERT(Expr, Msg)   \
  {                                \
    compile_time_error<bool(Expr)> \
      ERROR_##_Msg;                \
    (void)ERROR_##_Msg;            \
  }
```

上面首先声明了一个 struct 模板，然后仅对 `true` 的情况进行了特化，产生了一个 struct 的定义。这样。如果遇到 `compile_time_error<false>` 的情况——也就是下面静态断言里的 `Expr` 不为真的情况——编译就会失败报错，因为 `compile_time_error<false>` 从来就没有被定义过。

“动态”多态和“静态”多态的对比

我前面描述了面向对象的“动态”多态，也描述了 C++ 里基于泛型编程的“静态”多态。需要看到的是，两者解决的实际上是不太一样的问题。“动态”多态解决的是运行时的行为变化——就如我前面提到的，选择了一个形状之后，再选择在某个地方绘制这个形状——这个是无法在编译时确定的。“静态”多态或者“泛型”——解决的是很不同的问题，让适用于不同类型的“同构”算法可以用同一套代码来实现，实际上强调的是对代码的复用。

C++ 里提供了很多标准算法，都一样只作出了基本的约定，然后对任何满足约定的类型都可以工作。以排序为例，C++ 里的标准 `sort` 算法（以两参数的重载为例）只要求：

参数满足随机访问迭代器的要求。

迭代器指向的对象之间可以使用 < 来比较大小，满足严格弱序关系。

迭代器指向的对象可以被移动。

它的性能超出 C 的 `qsort`，因为编译器可以内联（inline）对象的比较操作；而在 C 里面比较只能通过一个额外的函数调用来实现。此外，C 的 `qsort` 函数要求数组指向的内容是可按比特复制的，C++ 的 `sort` 则要求迭代器指向的内容是可移动的，可适用于更广的情况。

C++ 里目前有大量这样的泛型算法。随便列举几个：

`sort`：排序

`reverse`：反转

`count`：计数

`find`：查找

`max`：最大值

`min`：最小值

`minmax`：最小值和最大值

`next_permutation`：下一个排列

`gcd`：最大公约数

`lcm`：最小公倍数

等等

内容小结

本讲我们对模板、泛型编程和静态多态做了最基本的描述，并和动态多态做了一定的比较。如果你不熟悉模板和泛型编程的话，应该在本讲之后已经对其有了初步的了解，我们可以在下面几讲中进行更深入的讨论。

课后思考

请你在课后读一下参考资料，了解一下各种不同的多态，然后想一想：

C++ 支持几种不同形式的多态？

为什么并非所有的语言都支持这些不同的多态方式？

欢迎你留言与我分享你的看法。

参考资料

[1] Wikipedia, “Polymorphism”. https://en.wikipedia.org/wiki/Polymorphism_(computer_science)

[1a] 维基百科, “多态”. https://zh.wikipedia.org/zh-cn/ 多型 _(计算机科学)

[2] Wikipedia, “Duck typing”. https://en.wikipedia.org/wiki/Duck_typing

[2a] 维基百科, “鸭子类型”. https://zh.wikipedia.org/zh-cn/ 鸭子类型

[3] Herb Sutter, “Why not specialize function templates?”. http://www.gotw.ca/publications/mill17.htm

上一篇　11 | Unicode：进入多文字支持的世界

精选留言 (2)

写留言

Geek_077da0
2019-12-23

老师您好，看到这一讲想问一个一直想问的问题。我是一个在校学生，目前学完了c++的基本语法知识并且看了一些相关的书籍，但平时能自己动手写代码的机会只有刷leetcode的时候，想请问一下老师，在去公司实习之前，有没有什么项目适合初学者练练手的。不然感觉自己看了这么多理论终究只是在纸上谈兵。

展开

1

tt
2019-12-23

从来没有把C++的模板编程和鸭子类型联系到一起，以前一提到鸭子类型，就想到了PYTHON和JAVASCRIPT。现在想想，按照鸭子类型的定义，那么JAVA也是支持它的。

一直感觉C++的模板编程就是一个静态实现的“动态类型子语言”：完全可以像写JAVASCRIPT一样写C++代码，只是需要先编译一下再运行。也许JAVASCRIPT的实现比如V8...

展开

作者回复: 静态语言的鸭子类型和动态语言还是有区别的。毕竟静态语言，如 C++，需要在编译时绑定所有的符号，否则就会出错……下面还会有例子，和 C++ 如何试图解决、改善这些问题的。编译期行为要讲上很多讲的。