Modern cpp 30 lectures — template

Monday, December 21, 2020 3:45 PM

面向对象和多态.

多态：用相同的代码得到不同结果. Eg. 以Shape类为例，可能会定义一些通用的功能，然后在子类进行实现或覆盖.

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

上面的类定义意味着所有的子类必须实现draw函数，可以认为是Shape定义了一个接口，在面向对象的设计里，接口抽象了一些基本行为，实现类里则去具体实现这些功能.

当我们有接口类的指针或引用时，实际可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，根据指针指向形状的不同，实际绘制出的可能是圆，三角或其他形状.

但这种面向对象的方式并不是唯一一种实现多态的方式，也可以不需要继承来实现circle, triangle等类，然后直接在这个类型的变量上调用draw方法. 唯一的要求只是这些不同的对象有"共通"的成员函数，这些成员函数有相同的名字和相同结构的参数，但并不要求参数类型相同.

容器类的共性

- 都有begin, end成员函数，所以容器不必继承一个共同的container基类，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，可以取得一个容器的大小，而不要求容器继承一个叫SizableContainer的基类.
- 很多容器有push_back成员函数，可以在尾部插入数据，同样，不需要一个叫BackPushableContainer的基类. push_back函数的参数显然是不一样的，但所有的push_back函数都只接收一个参数

虽然，C++的标准容器没有继承关系，但彼此之间有很多同构性. 这些同构性很难用继承体系来表达，也完全不必要用继承. C++的模板就足够.

C++模板

定义模板

最大公约数的辗转相除法.

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

问题：C++的整数类型不止int一种，为了让这个算法对长整型也有效，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，基本上是把int替换成了模板参数E，并在函数开头加了模板声明. 我们对于整数的要求，实际是：

- 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.

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)的时候才做.

对于上面的my_gcd，如果提供的是一般的整数类型，那是不会有问题的，但如果提供一些其他类型，就有可能出问题.

Eg. 以CLN，一个高精度数字库为例，如果使用它的cl_I高精度整数类型来调用my_gcd，出错信息为：

no match for 'operator%'

其原因是，虽然它的整数类 cl_I设计的很像普通的整数，但这个类的对象不支持%运算符.

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

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

那出错信息就会显示要求实例化的位置.

如果在显示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在其他地方实例化，从而不再产生其定义.

类似的，当我们在使用vector<int>这样的表达式时，就在隐式地实例化vector<int>，我们也可以用template class vector<int>来显式实例化，或者用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.

特化模板

如果遇到CLN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，怎么办？

- 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均可：

一. 添加对operator%的实现.

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

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.

为了通用，不直接使用cl_I的mod函数，而用my_mod把my_gcd改造：

```
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;
}

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

针对cl_I类，可以重载(overload)：

```
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(specialization).

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

特化相对于重载，是一种更通用的技巧，最主要原因是特化可用在类模板和函数模板上，而重载只能用于函数.

建议：对函数使用重载，对类模板进行特化.

展示特化更好的例子是C++11之前的静态断言，可大致实现static_assert：

```
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，编译器就会失败报错，因为compile_time_error<false>没有被定义过.

动态多态和静态多态的对比.

- 动态多态解决的是运行时的行为变化，Eg. 选择一个形状后，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，让适用于不同类型的同构算法可以用同一套代码实现，强调的是代码的复用.

Eg. C++提供了很多标准算法，都只作了基本约定，然后对任何满足约定的类型都可以工作. 以排序为例，sort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.