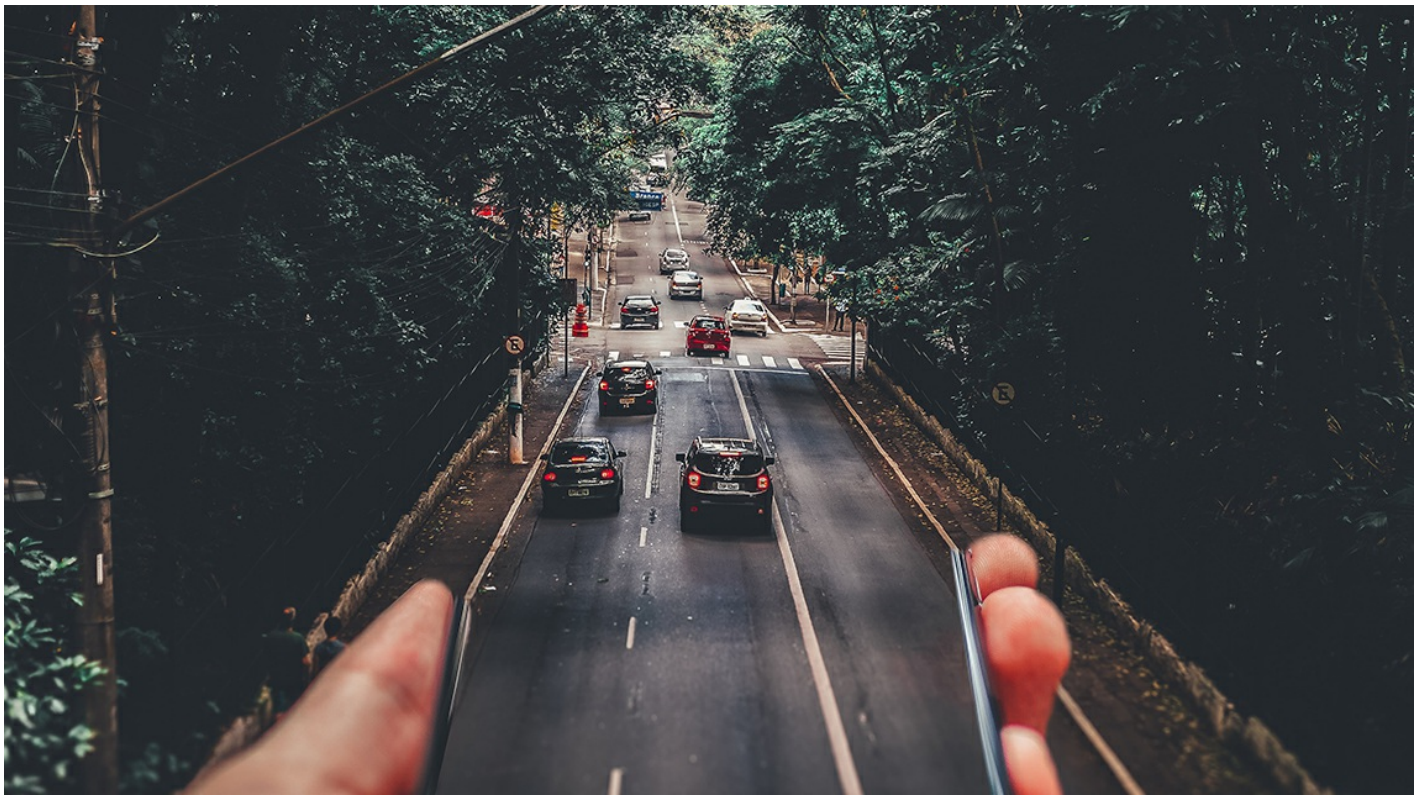
09讲易用性改进II：字面量、静态断言和成员函数说明符



你好，我是吴咏炜。

本讲我们继续易用性的话题，看看现代 C++ 带来的其他易用性改进。

自定义字面量

字面量（literal）是指在源代码中写出的固定常量，它们在 C++98 里只能是原生类型，如：

- `"hello"`，字符串字面量，类型是 `const char[6]`
- `1`，整数字面量，类型是 `int`
- `0.0`，浮点数字面量，类型是 `double`
- `3.14f`，浮点数字面量，类型是 `float`
- `123456789ul`，无符号长整数字面量，类型是 `unsigned long`

C++11 引入了自定义字面量，可以使用 `operator""` 后缀 来将用户提供的字面量转换成实际的类型。C++14 则在标准库中加入了不少标准字面量。下面这个程序展示了它们的用法：

```
#include <chrono>
#include <complex>
#include <iostream>
#include <string>
#include <thread>

using namespace std;

int main()
{
  cout << "i * i = " << 1i * 1i
       << endl;
  cout << "Waiting for 500ms"
       << endl;
  this_thread::sleep_for(500ms);
  cout << "Hello world"s.substr(0, 5)
       << endl;
}
```

输出是：

```
i * i = (-1,0)
Waiting for 500ms
Hello
```

上面这个例子展示了 C++ 标准里提供的帮助生成虚数、时间和 `basic_string` 字面量的后缀。一个需要注意的地方是，我在上面使用了 `using namespace std`，这会同时引入 `std` 名空间和里面的内联名空间（inline namespace），包括了上面的字面量运算符所在的三个名空间：

- `std::literals::complex_literals`
- `std::literals::chrono_literals`
- `std::literals::string_literals`

在产品项目中，一般不会（也不应该）全局使用 `using namespace std`（不过，为节约篇幅起见，专栏里的很多例子，特别是不完整的例子，还是默认使用了 `using namespace std`）。这种情况下，应当在使用到这些字面量的作用域里导入需要的名空间，以免发生冲突。在类似上面的例子里，就是在函数体的开头写：

```
using namespace std::literals::
  chrono_literals;
```

等等。

要在自己的类里支持字面量也相当容易，唯一的限制是非标准的字面量后缀必须以下划线 _ 打头。比如，假如我们有下面的长度类：

```
struct length {
  double value;
  enum unit {
    metre,
    kilometre,
    millimetre,
    centimetre,
    inch,
    foot,
    yard,
    mile,
  };
  static constexpr double factors[] =
    {1.0,    1000.0,  1e-3,
     1e-2,   0.0254,  0.3048,
     0.9144, 1609.344};
  explicit length(double v,
                  unit u = metre)
  {
    value = v * factors[u];
  }
};

length operator+(length lhs,
                 length rhs)
{
  return length(lhs.value +
                rhs.value);
}

// 可能有其他运算符
```

我们可以手写 `length(1.0, length::metre)` 这样的表达式，但估计大部分开发人员都不愿意这么做吧。反过来，如果我们让开发人员这么写，大家应该还是基本乐意的：

```
1.0_m + 10.0_cm
```

要允许上面这个表达式，我们只需要提供下面的运算符即可：

```
length operator"" _m(long double v)
{
  return length(v, length::metre);
}

length operator"" _cm(long double v)
{
  return length(v, length::centimetre);
}
```

如果美国国家航空航天局采用了类似的系统的话，火星气候探测者号的事故也许就不会发生了 [1]。当然，历史无法重来，而且 C++ 引入这样的语法已经是在事故发生之后十多年了……

关于自定义字面量的进一步技术细节，请参阅参考资料 [2]。

二进制字面量

你一定知道 C++ 里有 `0x` 前缀，可以让开发人员直接写出像 `0xFF` 这样的十六进制字面量。另外一个目前使用得稍少的前缀就是 `0` 后面直接跟 0–7 的数字，表示八进制的字面量，在跟文件系统打交道的时候还会经常用到：有经验的 Unix 程序员可能会觉得 `chmod(path, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)` 并不比 `chmod(path, 0644)` 更为直观。从 C++14 开始，我们对于二进制也有了直接的字面量：

```
unsigned mask = 0b111000000;
```

这在需要比特级操作等场合还是非常有用的。

不过，遗憾的是， I/O streams 里只有 `dec`、`hex`、`oct` 三个操纵器（manipulator），而没有 `bin`，因而输出一个二进制数不能像十进制、十六进制、八进制那么直接。一个间接方式是使用 `bitset`，但调用者需要手工指定二进制位数：

```
#include <bitset>
cout << bitset<9>(mask) << endl;
```

```
111000000
```

数字分隔符

数字长了之后，看清位数就变得麻烦了。有了二进制字面量，这个问题变得分外明显。C++14 开始，允许在数字型字面量中任意添加 `'` 来使其更可读。具体怎么添加，完全由程序员根据实际情况进行约定。某些常见的情况可能会是：

- 十进制数字使用三位的分隔，对应英文习惯的 thousand、million 等单位。
- 十进制数字使用四位的分隔，对应中文习惯的万、亿等单位。
- 十六进制数字使用两位或四位的分隔，对应字节或双字节。
- 二进制数字使用三位的分隔，对应文件系统的权限分组。
- 等等。

一些实际例子如下：

```
unsigned mask = 0b111'000'000;
long r_earth_equatorial = 6'378'137;
double pi = 3.14159'26535'89793;
const unsigned magic = 0x44'42'47'4E;
```

静态断言

C++98 的 `assert` 允许在运行时检查一个函数的前置条件是否成立。没有一种方法允许开发人员在编译的时候检查假设是否成立。比如，如果模板有个参数 `alignment`，表示对齐，那我们最好在编译时就检查 `alignment` 是不是二的整数次幂。之前人们用了一些模板技巧来达到这个目的，但输出的信息并不那么友善。比如，我之前使用的方法，会产生类似下面这样的输出：

```
In file included from test.cpp:1:
test.cpp: In instantiation of 'Something<alignment>::Something() [with i
nt alignment = 5]':
test.cpp:16:18:   required from here
test.cpp:9:53: error: static assertion failed: Alignment_must_be_power_o
f_two
         STATIC_ASSERT((alignment & (alignment - 1)) == 0,
                       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~
nvwa/static_assert.h:43:50: note: in definition of macro 'STATIC_ASSERT'
 #define STATIC_ASSERT(_Expr, _Msg) static_assert(_Expr, #_Msg)
                                                  ^~~~~
compilation terminated due to -Wfatal-errors.
```

能起作用，但不够直观。C++11 直接从语言层面提供了静态断言机制，不仅能输出更好的信息，而且适用性也更好，可以直接放在类的定义中，而不像之前用的特殊技巧只能放在函数体里。对于类似上面的情况，现在的输出是：

```
test.cpp: In instantiation of 'struct Something<5>':
test.cpp:16:18:   required from here
test.cpp:5:49: error: static assertion failed: Alignment must be power o
f two
     static_assert((alignment & (alignment - 1)) == 0,
                   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~
compilation terminated due to -Wfatal-errors.
```

静态断言语法上非常简单，就是：

```
static_assert(编译期条件表达式,
              可选输出信息);
```

产生上面的示例错误信息的代码是：

```
static_assert((alignment & (alignment - 1)) == 0,
  "Alignment must be power of two");
```

default 和 delete 成员函数

在类的定义时，C++ 有一些规则决定是否生成默认的特殊成员函数。这些特殊成员函数可能包括：

- 默认构造函数
- 析构函数
- 拷贝构造函数
- 拷贝赋值函数
- 移动构造函数
- 移动赋值函数

生成这些特殊成员函数（或不生成）的规则比较复杂，感兴趣的话你可以查看参考资料 [3]。每个特殊成员函数有几种不同的状态：

- 隐式声明还是用户声明
- 默认提供还是用户提供
- 正常状态还是删除状态

这三个状态是可组合的，虽然不是所有的组合都有效。隐式声明的必然是默认提供的；默认提供的才可能被删除；用户提供的也必然是用户声明的。

如果成员和父类没有特殊原因导致对象不可拷贝或移动，在用户不声明这些成员函数的情况下，编译器会自动产生这些成员函数，即隐式声明、默认提供、正常状态。有特殊成员、用户声明的话，情况就非常复杂了：

- 没有初始化的非静态 const 数据成员和引用类型数据成员会导致默认提供的默认构造函数被删除。
- 非静态的 const 数据成员和引用类型数据成员会导致默认提供的拷贝构造函数、拷贝赋值函数、移动构造函数和移动赋值函数被删除。
- 用户如果没有自己提供一个拷贝构造函数（必须形如 `Obj(Obj&)` 或 `Obj(const Obj&)`；不是模板），编译器会隐式声明一个。
- 用户如果没有自己提供一个拷贝赋值函数（必须形如 `Obj& operator=(Obj&)` 或 `Obj& operator=(const Obj&)`；不是模板），编译器会隐式声明一个。
- 用户如果自己声明了一个移动构造函数或移动赋值函数，则默认提供的拷贝构造函数和拷贝赋值函数被删除。
- 用户如果没有自己声明拷贝构造函数、拷贝赋值函数、移动赋值函数和析构函数，编译器会隐式声明一个移动构造函数。
- 用户如果没有自己声明拷贝构造函数、拷贝赋值函数、移动构造函数和析构函数，编译器会隐式声明一个移动赋值函数。
- ……

我不鼓励你去死记硬背这些规则，而是希望你在项目和测试中体会其缘由。我认为这些规则还相当合理，虽然有略偏保守之嫌。尤其是关于移动构造和赋值：只要用户声明了另外的特殊成员函数中的任何一个，编译器就不默认提供了。不过嘛，缺省慢点总比缺省不安全要好……

我们这儿主要要说的是，我们可以改变缺省行为，在编译器能默认提供特殊成员函数时将其删除，或在编译器不默认提供特殊成员函数时明确声明其需要默认提供（不过，要注意，即使用户要求默认提供，编译器也可能根据其他规则将特殊成员函数标为删除）。

还是举例子来说明一下。对于下面这样的类，编译器看到有用户提供的构造函数，就会不默认提供默认构造函数：

```
template <typename T>
class my_array {
public:
  my_array(size_t size);
  …
private:
  T*     data_{nullptr};
  size_t size_{0};
};
```

在没有默认初始化时，我们如果需要默认构造函数，就需要手工写一个，如：

```
  my_array()
    : data_(nullptr)
    , size_(0) {}
```

可有了默认初始化之后，这个构造函数显然就不必要了，所以我们现在可以写：

```
  my_array() = default;
```

再来一个反向的例子。我们 [第 1 讲] 里的 `shape_wrapper`，它的复制行为是不安全的。我们可以像 [第 2 讲] 里一样去改进它，但如果正常情况不需要复制行为、只是想防止其他开发人员误操作时，我们可以简单地在类的定义中加入：

```
class shape_wrapper {
  …
  shape_wrapper(
    const shape_wrapper&) = delete;
  shape_wrapper& operator=(
    const shape_wrapper&) = delete;
  …
};
```

在 C++11 之前，我们可能会用在 `private` 段里声明这些成员函数的方法，来达到相似的目的。但目前这个语法效果更好，可以产生更明确的错误信息。另外，你可以注意一下，用户声明成删除也是一种声明，因此编译器不会提供默认版本的移动构造和移动赋值函数。

override 和 final 说明符

`override` 和 `final` 是两个 C++11 引入的新说明符。它们不是关键词，仅在出现在函数声明尾部时起作用，不影响我们使用这两个词作变量名等其他用途。这两个说明符可以单个或组合使用，都是加在类成员函数声明的尾部。

`override` 显式声明了成员函数是一个虚函数且覆盖了基类中的该函数。如果有 `override` 声明的函数不是虚函数，或基类中不存在这个虚函数，编译器会报告错误。这个说明符的主要作用有两个：

- 给开发人员更明确的提示，这个函数覆写了基类的成员函数；
- 让编译器进行额外的检查，防止程序员由于拼写错误或代码改动没有让基类和派生类中的成员函数名称完全一致。

`final` 则声明了成员函数是一个虚函数，且该虚函数不可在派生类中被覆盖。如果有一点没有得到满足的话，编译器就会报错。

`final` 还有一个作用是标志某个类或结构不可被派生。同样，这时应将其放在被定义的类或结构名后面。

用法示意如下：

```
class A {
public:
  virtual void foo();
  virtual void bar();
  void foobar();
};

class B : public A {
public:
  void foo() override; // OK
  void bar() override final; // OK
  //void foobar() override;
  // 非虚函数不能 override
};

class C final : public B {
public:
  void foo() override; // OK
  //void bar() override;
  // final 函数不可 override
};

class D : public C {
  // 错误：final 类不可派生
  …
};
```

内容小结

今天我们介绍了现代 C++ 引入的另外几个易用性改进：自定义字面量，二进制字面量，数字分隔符，静态断言，default 和 delete 成员函数，及 override 和 final。同上一讲介绍的易用性改进一样，这些新功能可以改进代码的可读性，同时也不会带来额外的开销。在任何有条件使用满足新 C++ 标准的编译器的项目中，都应该考虑使用这些新特性。

课后思考

你最喜欢的 C++ 易用性改进是什么？为什么？

欢迎留言和我分享你的看法！

参考资料

[1] Wikipedia, “Mars Climate Orbiter”. https://en.wikipedia.org/wiki/Mars_Climate_Orbiter

[1a] 维基百科, “火星气候探测者号”. https://zh.wikipedia.org/zh-cn/火星氣候探測者號

[2] cppreference.com, “User-defined literals”. https://en.cppreference.com/w/cpp/language/user_literal

[2a] cppreference.com, “用户定义字面量”. https://zh.cppreference.com/w/cpp/language/user_literal

[3] cppreference.com, “Non-static member functions”, section “Special member functions”. https://en.cppreference.com/w/cpp/language/member_functions

[3a] cppreference.com, “非静态成员函数”, “特殊成员函数”部分. https://zh.cppreference.com/w/cpp/language/member_functions

精选留言

三味

如果对一个函数声明了final, 我觉得没有必要在添加override了吧.
override就是明确声明这是一个继承来的函数, final同样也是这个意思, 只不过final更霸道, 后续的不要在继承了!
如果从一开始就不想让别的函数去继承而写final, 那就根本没必要去virtual它. 何必要在虚函数表中添加一个没有继承作用的虚函数呢?

PS: 数字分隔符和自定义字面量真是学到了. 在9102年的尾巴, 我才知道有这么邪道的用法...
PPS: 前些日子看了一些国外游戏大牛各种喷C++的帖子. 本章的所有内容应该都是他们喷的范围吧... 那些人特别看中编译时间, 追求极致的运行效率... 有个人专门对比了int a=7; 和 int a{7}的性能差别...从编译到运行时间... 利用宏展开的方式, 对这两个例子分别做了百万次展开, 如果用vs测试都能爆IDE内存的级别... 我觉得对于中小型对性能不是特别敏感的程序, 这些还是很有用的.
PPPS: 最近对Data-oriented design感兴趣, 不知道作者以后是否有开这类理论实战课程的计划捏? 我搜索上面喷神, 就是从这里开始搜索出来的...

2019-12-20 10:50

作者回复

对，final override 合法但不必要。

当然不是所有内容都是被那个游戏开发人士喷的。他抱怨的是会导致编译速度下降，以及非优化编译性能差的那部份功能。通常都是模板相关的。所以这一讲的内容不属于其中。

2019-12-20 13:22

小一日一

我最喜欢的C++易用性改进及理由：
auto： 少打字
scope for：少打字
类成员默认补始化语法：少打字
default 和 delete 成员函数：简化对类行为的控制难度
自定义字面量：代码看起来舒服。

2019-12-16 11:48

作者回复

都用上了吗？很好啊。

2019-12-16 13:34

中年男子

既然有了这些特性， 我觉的就得在平时开发中用起来，不用就没有用，完全浪费了大神的研究

2019-12-24 15:07

作者回复

对的，一定要用起来！

2019-12-24 18:15

木瓜777

您好，您有没有感觉比较好的开源c++项目推荐？

希望从别人的项目中学到一些经验，谢谢！

2019-12-17 20:42

作者回复

C++ 项目入门都不容易，要找自己有兴趣的领域是关键。除了第 6 讲评论里推荐的那些，可以考虑下面两个（我将来也会讲到）：

- EasyLogging++
- Catch2

2019-12-18 07:32

hello world

平时自己主要用到的就是auto和default和delete override，方便且避免出错

2019-12-16 18:27

作者回复

其他的也要考虑用起来啊。

2019-12-16 20:50

墨梵

auto 和 for loop range base的搭配

2019-12-16 09:21

作者回复

嗯，这个搭配很爽的。

2019-12-16 13:28

西钾钾

auto 与 新的 for 遍历方法，因为实在是方便而且用的最多。其他的特性 ... 工作中貌似还没怎么用过

2019-12-16 08:17

作者回复

没成本的改进，能用就用起来吧。

2019-12-16 09:16