15 | constexpr：一个常态的世界

2019-12-30 吴咏炜

现代C++实战30讲 进入课程 >

讲述：吴咏炜
时长 16:52　大小 15.46M

你好，我是吴咏炜。

我们已经连续讲了几讲比较累人的编译期编程了。今天我们还是继续这个话题，但是，相信今天学完之后，你会感觉比之前几讲要轻松很多。C++ 语言里的很多改进，让我们做编译期编程也变得越来越简单了。

初识 constexpr

我们先来看一些例子：

复制代码

```
1 int sqr(int n)
```

```
{
  return n * n;
}

int main()
{
  int a[sqr(3)];
}
```

想一想，这个代码合法吗？

看过之后，再想想这个代码如何？

复制代码

```
int sqr(int n)
{
  return n * n;
}

int main()
{
  const int n = sqr(3);
  int a[sqr(3)];
}
```

还有这个？

复制代码

```
#include <array>

int sqr(int n)
{
  return n * n;
}

int main()
{
  std::array<int, sqr(3)> a;
}
```

此外，我们前面模板元编程里的那些类里的 `static const int` 什么的，你认为它们能用在上面的几种情况下吗？

如果以上问题你都知道正确的答案，那恭喜你，你对 C++ 的理解已经到了一个不错的层次了。但问题依然在那里：这些问题的答案不直观。并且，我们需要一个比模板元编程更方便的进行编译期计算的方法。

在 C++11 引入、在 C++14 得到大幅改进的 `constexpr` 关键字就是为了解决这些问题而诞生的。它的字面意思是 constant expression，常量表达式。存在两类 `constexpr` 对象：

- `constexpr` 变量（唉……😟）
- `constexpr` 函数

一个 `constexpr` 变量是一个编译时完全确定的常数。一个 `constexpr` 函数至少对于某一组实参可以在编译期间产生一个编译期常数。

注意一个 `constexpr` 函数不保证在所有情况下都会产生一个编译期常数（因而也是可以作为普通函数来使用的）。编译器也没法通用地检查这点。编译器唯一强制的是：

- `constexpr` 变量必须立即初始化
- 初始化只能使用字面量或常量表达式，后者不允许调用任何非 `constexpr` 函数

`constexpr` 的实际规则当然稍微更复杂些，而且随着 C++ 标准的演进也有着一些变化，特别是对 `constexpr` 函数如何实现的要求在慢慢放宽。要了解具体情况包括其在不同 C++ 标准中的限制，可以查看参考资料 [1]。下面我们也会回到这个问题略作展开。

拿 `constexpr` 来改造开头的例子，下面的代码就完全可以工作了：

复制代码

```
#include <array>

constexpr int sqr(int n)
{
```

```
  return n * n;
}

int main()
{
  constexpr int n = sqr(3);
  std::array<int, n> a;
  int b[n];
}
```

要检验一个 `constexpr` 函数能不能产生一个真正的编译期常量，可以把结果赋给一个 `constexpr` 变量。成功的话，我们就确认了，至少在这种调用情况下，我们能真正得到一个编译期常量。

constexpr 和编译期计算

上面这些当然有点用。但如果只有这点用的话，就不值得我专门来写一讲了。更强大的地方在于，使用编译期常量，就跟我们之前的那些类模板里的 `static const int` 变量一样，是可以进行编译期计算的。

以 [第 13 讲] 提到的阶乘函数为例，和那个版本基本等价的写法是：

复制代码

```
constexpr int factorial(int n)
{
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

然后，我们用下面的代码可以验证我们确实得到了一个编译期常量：

复制代码

```
int main()
{
  constexpr int n = factorial(10);
  printf("%d\n", n);
}
```

编译可以通过，同时，如果我们看产生的汇编代码的话，一样可以直接看到常量3628800。

这里有一个问题：在这个 `constexpr` 函数里，是不能写 `static_assert(n >= 0)` 的。一个 `constexpr` 函数仍然可以作为普通函数使用——显然，传入一个普通 `int` 是不能使用静态断言的。替换方法是在 `factorial` 的实现开头加入：

```
  if (n < 0) {
    throw std::invalid_argument(
      "Arg must be non-negative");
  }
```

如果你在 `main` 里写 `constexpr int n = factorial(-1);` 的话，就会看到编译器报告抛出异常导致无法得到一个常量表达式。建议你自己尝试一下。

constexpr 和 const

初学 `constexpr` 时，一个很可能有的困惑是，它跟 `const` 用法上的区别到底是什么。产生这种困惑是正常的，毕竟 `const` 是个重载了很多不同含义的关键字。

`const` 的原本和基础的含义，自然是表示它修饰的内容不会变化，如：

```
const int n = 1:
n = 2;  //  出错!
```

注意 `const` 在类型声明的不同位置会产生不同的结果。对于常见的 `const char*` 这样的类型声明，意义和 `char const*` 相同，是指向常字符的指针，指针指向的内容不可更改；但和 `char * const` 不同，那代表指向字符的常指针，指针本身不可更改。本质上，`const` 用来表示一个**运行时常量**。

在 C++ 里，`const` 后面渐渐带上了现在的 `constexpr` 用法，也代表**编译期常数**。现在——在有了 `constexpr` 之后——我们应该使用 `constexpr` 在这些用法中替换 `const` 了。从编译器的角度，为了向后兼容性，`const` 和 `constexpr` 在很多情况下还是等价的。但有时候，它们也有些细微的区别，其中之一为是否内联的问题。

内联变量

C++17 引入了内联（inline）变量的概念，允许在头文件中定义内联变量，然后像内联函数一样，只要所有的定义都相同，那变量的定义出现多次也没有关系。对于类的静态数据成员，`const` 缺省是不内联的，而 `constexpr` 缺省就是内联的。这种区别在你用 `&` 去取一个 `const int` 值的地址、或将其传到一个形参类型为 `const int&` 的函数去的时候（这在 C++ 文档里的行话叫 ODR-use），就会体现出来。

下面是个合法的完整程序：

复制代码

```
#include <iostream>

struct magic {
  static const int number = 42;
};

int main()
{
  std::cout << magic::number
            << std::endl;
}
```

我们稍微改一点：

复制代码

```
#include <iostream>
#include <vector>

struct magic {
  static const int number = 42;
};

int main()
{
```

```
10    std::vector<int> v;
11    //  调用  push_back(const T&)
12    v.push_back(magic::number);
13    std::cout << v[0] << std::endl;
14  }
```

程序在链接时就会报错了，说找不到 `magic::number`（注意：MSVC 缺省不报错，但使用标准模式——`/Za` 命令行选项——也会出现这个问题）。这是因为 ODR-use 的类静态常量也需要有一个定义，在没有内联变量之前需要在某一个源代码文件（非头文件）中这样写：

复制代码

```
1  const int magic::number = 42;
```

必须正正好好一个，多了少了都不行，所以叫 one definition rule。内联函数，现在又有了内联变量，以及模板，则不受这条规则限制。

修正这个问题的简单方法是把 `magic` 里的 `static const` 改成 `static constexpr` 或 `static inline const`。前者可行的原因是，类的静态 constexpr 成员变量默认就是内联的。const 常量和类外面的 constexpr 变量不默认内联，需要手工加 `inline` 关键字才会变成内联。

constexpr 变量模板

变量模板是 C++14 引入的新概念。之前我们需要用类静态数据成员来表达的东西，使用变量模板可以更简洁地表达。`constexpr` 很合适用在变量模板里，表达一个和某个类型相关的编译期常量。由此，type traits 都获得了一种更简单的表示方式。再看一下我们在 [第 13 讲] 用过的例子：

复制代码

```
1  template <class T>
2  inline constexpr bool
3    is_trivially_destructible_v =
4      is_trivially_destructible<
5        T>::value;
```

了解了变量也可以是模板之后，上面这个代码就很容易看懂了吧？这只是一个小小的语法糖，允许我们把 `is_trivially_destructible<T>::value` 写成 `is_trivially_destructible_v<T>`。

constexpr 变量仍是 const

一个 `constexpr` 变量仍然是 const 常类型。需要注意的是，就像 `const char*` 类型是指向常量的指针、自身不是 const 常量一样，下面这个表达式里的 `const` 也是不能缺少的：

复制代码

```
1 constexpr int a = 42;
2 constexpr const int& b = a;
```

第二行里，`constexpr` 表示 `b` 是一个编译期常量，`const` 表示这个引用是常量引用。去掉这个 `const` 的话，编译器就会认为你是试图将一个普通引用绑定到一个常数上，报一个类似下面的错误信息：

> **error:** binding reference of type ‘**int&**’ to ‘**const int**’ discards qualifiers

如果按照 const 位置的规则，`constexpr const int& b` 实际该写成 `const int& constexpr b`。不过，`constexpr` 不需要像 `const` 一样有复杂的组合，因此永远是写在类型前面的。

constexpr 构造函数和字面类型

一个合理的 `constexpr` 函数，应当至少对于某一组编译期常量的输入，能得到编译期常量的结果。为此，对这个函数也是有些限制的：

最早，`constexpr` 函数里连循环都不能有，但在 C++14 放开了。

目前，`constexpr` 函数仍不能有 `try … catch` 语句和 `asm` 声明，但到 C++20 会放开。

`constexpr` 函数里不能使用 `goto` 语句。

等等。

一个有意思的情况是一个类的构造函数。如果一个类的构造函数里面只包含常量表达式、满足对 `constexpr` 函数的限制的话（这也意味着，里面不可以有任何动态内存分配），并且类的析构函数是平凡的，那这个类就可以被称为是一个字面类型。换一个角度想，对 `constexpr` 函数——包括字面类型构造函数——的要求是，得让编译器能在编译期进行计算，而不会产生任何“副作用”，比如内存分配、输入、输出等等。

为了全面支持编译期计算，C++14 开始，很多标准类的构造函数和成员函数已经被标为 `constexpr`，以便在编译期使用。当然，大部分的容器类，因为用到了动态内存分配，不能成为字面类型。下面这些不使用动态内存分配的字面类型则可以在常量表达式中使用：

- array
- initializer_list
- pair
- tuple
- string_view
- optional
- variant
- bitset
- complex
- chrono::duration
- chrono::time_point
- `shared_ptr`（仅限默认构造和空指针构造）
- `unique_ptr`（仅限默认构造和空指针构造）
- …

下面这个玩具例子，可以展示上面的若干类及其成员函数的行为：

```cpp
#include <array>
#include <iostream>
#include <memory>
#include <string_view>

using namespace std;

int main()
{
  constexpr string_view sv{"hi"};
  constexpr pair pr{sv[0], sv[1]};
  constexpr array a{pr.first, pr.second};
  constexpr int n1 = a[0];
  constexpr int n2 = a[1];
  cout << n1 << ' ' << n2 << '\n';
}
```

复制代码

编译器可以在编译期即决定 `n1` 和 `n2` 的数值；从最后结果的角度，上面程序就是输出了两个整数而已。

if constexpr

上一讲的结尾，我们给出了一个在类型参数 `C` 没有 `reserve` 成员函数时不能编译的代码：

```cpp
template <typename C, typename T>
void append(C& container, T* ptr,
            size_t size)
{
  if (has_reserve<C>::value) {
    container.reserve(
      container.size() + size);
  }
  for (size_t i = 0; i < size;
       ++i) {
    container.push_back(ptr[i]);
  }
}
```

复制代码

在 C++17 里，我们只要在 `if` 后面加上 `constexpr`，代码就能工作了 [2]。当然，它要求括号里的条件是个编译期常量。满足这个条件后，标签分发、`enable_if` 那些技巧就不那么有用了。显然，使用 `if constexpr` 能比使用其他那些方式，写出更可读的代码……

output_container.h 解读

到了今天，我们终于把 output_container.h（[3]）用到的 C++ 语法特性都讲过了，我们就拿里面的代码来讲解一下，让你加深对这些特性的理解。

复制代码

```
// Type trait to detect std::pair
template <typename T>
struct is_pair : std::false_type {};
template <typename T, typename U>
struct is_pair<std::pair<T, U>>
  : std::true_type {};
template <typename T>
inline constexpr bool is_pair_v =
  is_pair<T>::value;
```

这段代码利用模板特化（[第 12 讲]、[第 14 讲]）和 `false_type`、`true_type` 类型（[第 13 讲]），定义了 `is_pair`，用来检测一个类型是不是 `pair`。随后，我们定义了内联 `constexpr` 变量（本讲）`is_pair_v`，用来简化表达。

复制代码

```
// Type trait to detect whether an
// output function already exists
template <typename T>
struct has_output_function {
  template <class U>
  static auto output(U* ptr)
    -> decltype(
      std::declval<std::ostream&>()
        << *ptr,
      std::true_type());
  template <class U>
  static std::false_type
  output(...);
  static constexpr bool value =
    decltype(
      output<T>(nullptr))::value;
};
template <typename T>
inline constexpr bool
  has_output_function_v =
    has_output_function<T>::value;
```

这段代码使用 SFINAE 技巧（[第 14 讲]），来检测模板参数 `T` 的对象是否已经可以直接输出到 `ostream`。然后，一样用一个内联 `constexpr` 变量来简化表达。

复制代码

```
// Output function for std::pair
template <typename T, typename U>
std::ostream& operator<<(
  std::ostream& os,
  const std::pair<T, U>& pr);
```

再然后我们声明了一个 `pair` 的输出函数（标准库没有提供这个功能）。我们这儿只是声明，是因为我们这儿有两个输出函数，且可能互相调用。所以，我们要先声明其中之一。

下面会看到，`pair` 的通用输出形式是“(x, y)”。

复制代码

```
// Element output function for
// containers that define a key_type
// and have its value type as
// std::pair
template <typename T, typename Cont>
auto output_element(
  std::ostream& os,
  const T& element, const Cont&,
  const std::true_type)
  -> decltype(
    std::declval<
      typename Cont::key_type>(),
    os);
// Element output function for other
// containers
template <typename T, typename Cont>
auto output_element(
  std::ostream& os,
  const T& element, const Cont&,
  ...) -> decltype(os);
```

对于容器成员的输出，我们也声明了两个不同的重载。我们的意图是，如果元素的类型是 `pair` 并且容器定义了一个 `key_type` 类型，我们就认为遇到了关联容器，输出形式为“x => y”（而不是“(x, y)”）。

复制代码

```
// Main output function, enabled
// only if no output function
// already exists
template <
  typename T,
  typename = std::enable_if_t<
    !has_output_function_v<T>>>
auto operator<<(std::ostream& os,
                const T& container)
  -> decltype(container.begin(),
              container.end(), os)
…
```

主输出函数的定义。注意这儿这个函数的启用有两个不同的 SFINAE 条件：

用 `decltype` 返回值的方式规定了被输出的类型必须有 `begin()` 和 `end()` 成员函数。

用 `enable_if_t` 规定了只在被输出的类型没有输出函数时才启用这个输出函数。否则，对于 `string` 这样的类型，编译器发现有两个可用的输出函数，就会导致编译出错。

我们可以看到，用 `decltype` 返回值的方式比较简单，不需要定义额外的模板。但表达否定的条件还是要靠 `enable_if`。此外，因为此处是需要避免有二义性的重载，constexpr 条件语句帮不了什么忙。

复制代码

```
  using element_type =
    decay_t<decltype(
      *container.begin())>;
  constexpr bool is_char_v =
    is_same_v<element_type, char>;
  if constexpr (!is_char_v) {
    os << "{ ";
  }
```

对非字符类型，我们在开始输出时，先输出“{ ”。这儿使用了 `decay_t`，是为了把类型里的引用和 const/volatile 修饰去掉，只剩下值类型。如果容器里的成员是 `char`，这儿会把 `char&` 和 `const char&` 还原成 `char`。

后面的代码就比较简单了。可能唯一需要留意的是下面这句：

复制代码

```
  output_element(
    os, *it, container,
    is_pair<element_type>());
```

这儿我们使用了标签分发技巧来输出容器里的元素。要记得，`output_element` 不纯粹使用标签分发，还会检查容器是否有 `key_type` 成员类型。

复制代码

```
template <typename T, typename Cont>
auto output_element(
  std::ostream& os,
  const T& element, const Cont&,
  const std::true_type)
  -> decltype(
    std::declval<
      typename Cont::key_type>(),
    os)
{
  os << element.first << " => "
     << element.second;
  return os;
}

template <typename T, typename Cont>
auto output_element(
  std::ostream& os,
  const T& element, const Cont&,
  ...) -> decltype(os)
{
  os << element;
  return os;
}
```

`output_element` 的两个重载的实现都非常简单，应该不需要解释了。

复制代码

```
template <typename T, typename U>
std::ostream& operator<<(
  std::ostream& os,
  const std::pair<T, U>& pr)
```

```
5 {
6   os << '(' << pr.first << ", "
7      << pr.second << ')';
8   return os;
9 }
```

同样，`pair` 的输出的实现也非常简单。

唯一需要留意的，是上面三个函数的输出内容可能还是容器，因此我们要将其实现放在后面，确保它能看到我们的通用输出函数。

要看一下用到 output_container 的例子，可以回顾 [第 4 讲] 和 [第 5 讲]。

内容小结

本讲我们介绍了编译期常量表达式和编译期条件语句，可以看到，这两种新特性对编译期编程有了很大的改进，可以让代码变得更直观。最后我们讨论了我们之前用到的容器输出函数 output_container 的实现，里面用到了多种我们目前讨论过的编译期编程技巧。

课后思考

请你仔细想一想：

1. 如果没有 constexpr 条件语句，这个容器输出函数需要怎样写？
2. 这种不使用 constexpr 的写法有什么样的缺点？推而广之，constexpr 条件语句的意义是什么？

参考资料

[1] cppreference.com, “constexpr specifier”. https://en.cppreference.com/w/cpp/language/constexpr

[1a] cppreference.com, “constexpr 说明符”. https://zh.cppreference.com/w/cpp/language/constexpr

[2] cppreference.com, “if statement”, section “constexpr if”.
https://en.cppreference.com/w/cpp/language/if

[2a] cppreference.com, “if 语句”, “constexpr if” 部分.
https://zh.cppreference.com/w/cpp/language/if

[3] 吴咏炜, output_container.
https://github.com/adah1972/output_container/blob/master/output_container.h

上一篇 14 | SFINAE：不是错误的替换失败是怎么回事?

精选留言 (3)

写留言

jerry.tan
2019-12-30

您好老师, 请问想学C++ 您有什么比较好的推荐的开发工具吗 谢谢

1

李亮亮
2019-12-30

我觉得我学习这个专栏只是为了能看懂这些新特性，写是写不出来，规则太多太复杂了。


hello world
2019-12-30

如果没有consrexpr条件语句那输出函数就应该写两个吧，也是用sfinae，用那种enable_if形式，true or false

展开