Effective Modern C++ 条款15 尽可能使用constexpr

尽可能使用constexpr

如果要选出C++11中最让人迷惑的新关键字，那么大概是**constexpr**。当**constexpr**用于对象时，它本质上就是加强版的**const**，但它用于**函数**时，它拥有不同的意思。**constexpr**再迷惑，也是值得的，因为当**constexpr**与你想要表达的一致时，你肯定会用它。

在概念上，**constexpr**表明一个值不仅是常量，还是在编译期间可知。这概念只是拼图的一部分，因为当**constexpr**用于函数时，有点微妙的区别。免得我破坏了最后的惊喜，我现在只可以说，你不能假定**constexpr**函数的返回结果是**const**的，也不能理所当然的人物它们的返回值在编译期间可知。可能会很有趣，这些特性。**constexpr**函数不需要返回**const**结果和编译器可知结果，这是有益的。

不过我们还是先讲**constexpr**对象，这些对象呢，事实上和**const**一样，它们的值在编译期间就知道了。

那些在编译期间就可知的值是享有特权的。例如，它们可能存放在只读的内存区域中，特别是为那些内嵌系统的开发者，这是一个相当重要的特性。在C++的上下文中需要一个整型常量表达式(integral constant expression)时，一个常量的和编译期间可知的整型数具有广泛适应性。这种上下文包括数组大小的表示，整型模板参数（包括**std::array**对象的长度），枚举的值，对齐说明，等等。如果你想要一个变量，用于例说的东西，那么你肯定想要把那个变量声明为**constexpr**，因为编译器会确保它在编译期间有值：

```
int sz;   // non-constexpr variable
...
constexpr auto arraySize1 = sz;   // 错误，编译期间不知道sz的值

std::array<int, sz> data1;   // 错误，同样的问题

constexpr auto arraySize2 =  10;  // 正确，10在编译期间是常量

std::array<int, arraySize2> data2;  // 正确，arraySize2是constexpr的
```

请注意**const**并不提供与**constexpr**相同的保证，因为**const**对象在编译时不需要用已知的值初始化：

```
int sz;  // 如前
...
const auto arraySize = sz;  // 正确，arraySize是sz的**const**拷贝

std::array<int, arraySize> data; // 错误，arraySize的值在编译期间不可知
```

简而言之，所有的constexpr对象都是const的，但并不是所有的const对象都是constexpr。仅有constexpr对象具备编译期间确其值的能力。

我们可以简单地认为，所有**constexpr**对象都是**const**的，但是不是所有的**const**对象都是**constexpr**的。如果你想要编译器保证变量编译期有值，即上下文请求了一个编译期间的常量，那么能用的工具是**constexpr**，而不是**const**。

当涉及到**constexpr**函数的时候，**constexpr**对象的使用会变得更加有趣。当编译期间的常量作为参数传递给**constexpr**函数时，这种函数会返回编译期间常量。如果函数的参数在运行期间才能知道，函数返回的也是运行时的值。听起来有点乱，正确的规则：

- **constexpr**函数可以用在需求编译期间常量的上下文。在这种上下文中，如果你传递参数的值在编译期间已知，那么函数的结果会在编译期间计算。如果任何一个参数的值在编译期间未知，代码将不能通过编译。
- 如果用一个或者多个在编译期间未知的值作为参数调用**constexpr**函数，函数的行为和普通的函数一样，在运行期间计算结果。这意味着你不需要用两个函数来表示这个操作——一个在编译期间和一个在运行期间。**constexpr**函数具有两个功能。

假设我们需要一个数据结构来保存某个实验的结果，这个实验可在不同的条件下进行。例如，在实验期间，光的强度可高可低，风速和温度也可变化。如果与实验有关的环境条件有n个，每个环境变量又有3种状态，那么就有3^n种情况。存储实验可能出现的所有结果，就要求数据结构有足够大的空间保存3^n个值。假设每个结果是int值，然后在编译期间已知（或者可计算），那么选择**std::array**这数据结构将会合情合理。C++标准库提供**std::pow**，是我们需要的数学计算函数，但这里会有两个问题。第一，**std::pow**作用于两个浮点型指针，而我们需要的是一个整型结果。第二，**std::pow**不是**constexpr**的，所以我们不能用它的结果来指定**std::array**的值。

幸运的是，我们可以自己写**pow**函数。等下我会展示它是怎么做的，但我们先看看它是怎样声明和使用的：

```
constexpr   // pow是个constexpr函数
int pow(int base, int exp) noexcept   // 函数不会抛出引出
{
    ...    // 实现看下面
}

constexpr auto numCouds = 5;   // 条件个数

std::array<int, pow(3, numCouds)> results;   // results引3^n个元素
```

pow前面的constexpr并不表明pow返回一个const值，它表示pow是一个constexpr函数，即：
① 如果base和exp是编译时常量，那么pow的结果可以用作编译时常量。
② 如果base和exp不是编译时常量，pow的结果将在运行时计算。
这意味着pow不仅用于在编译期计算std::array的大小，还可以在运行时环境中调用它：

```
auto base = readFromDB("base");     // 在运行期间
auto exp = readFromDB("exponent");  // 获取值

auto baseToExp = pow(base, exp);   // 在运行期间调用pow
```

因为用编译期间的值作为参数调用**constexpr**函数一定要返回编译期间的结果，所以会有限制强加于它们的实现。C++11和C++14的限制不同。

在C++11，**constexpr**只能有一个**return**语句。听起来不是什么限制，因为可以用两个技巧。第一个是`?：`运算符代替if-else语句，第二个是可以用递归。所以**pow**可以这样实现：

```
constexpr int pow(int base, int exp) noexcept
{
    return (exp == 0 ? 1 : base * pow(base, exp - 1));
}
```

这可以运行，但是很难想象除了大神还有谁能把它写得这么好。在C++14中，**constexpr**函数的限制大幅宽松，所以这种函数实现成为可能：

```
constexpr int por(int base, int exp) noexcept
{                              base^exp
    auto result = 1;
    for (int i=0; i < exp; ++i) result *= base;
    return results;
};
```

constexpr函数限制持有和返回的类型为字面值类型（literal type），本质上就是一些在编译期间可确定值的类型。在C++中，除了**void**之外的内置类型都是字面值类型，不过用户定义的类型也有可能是字面值类型，因为构造函数和其他成员函数可能是**constexpr**的：

```
class Point {
public:
    constexpr Point(double xVal = 0, double yVal = 0) noexcept
    : x(xVal), y(yVal)
    {}

    constexpr double xValue() const noexcept { return xVal; }
    constexpr double yValue() const noexcept { return yVal; }

    void setX(double newX) noexcept { x = newX; }
    void setY(double newY) noexcept { y = newY; }

private:
    double x, y;
};
```

在这里，Point的构造函数可以被声明为**constexpr**，因为如果传进来的参数在编译时就可以知道，那么由P构造的成员变量的值在编译时也可以被知道。因此Point可以用**constexpr**初始化：

```
constexpr Point p1(9.4, 27.7);  // 正确，在编译时“运行”constexpr构造

constexpr Point p2(28.8, 5.3);  // 也正确
```

同样的，获取函数（getter）xValue和yValue也可以是**constexpr**，因为如果它们被一个编译期间已知的Point对象调用（例如，一个**constexpr**的Point对象），成员变量x和y的值在编译时是已知的，这使一个**constexpr**函数调用Point的获取函数并用其结果来初始化一个**constexpr**对象成为可能：

```
constexpr
Point midpoint(const Point &p1, const Point &p2) noexcept
{
    return { (p1.xValue + p2.xValue)) / 2,       // 调用constexpr
             (p1.yValue + p2.yValue)) / 2 };     // 成员函数
}

constexpr auto mid = midpoint(p1, p2);  // 用**constexpr**函数的结果                    // 初始化constexpr对象。
```

这很亦可赛艇，这意味着对象mid的初始化涉及到构造函数、获取函数、非成员函数的调用，然后创建在只读内存区域！这意味着你可以将一个类似 `mid.xValue() * 10` 的表达式用于模板参数或者一个指定枚举值的表达式！这意味着传统意义上，编译期需完成的工作与运行期间需完成的工作之间的严格清晰的线变模糊了，而一些传统意义上运行时的工作可以迁移到编译期。参与迁移的代码越多，软件运行得越快（但是，编译的时间可能变长）。

在C++11，有两个限制因素妨碍把Point的成员变量setX和setY声明为**constexpr**。第一，它们改变了它们操作的值，然后在C++11，**constexpr**成员函数是隐式声明为**const**的。第二，它们的返回类型是**void**，然后在C++11，**void**不是字面值类型。都是这两个限制在C++14被解除了，所以在C++14，设置函数（setter）也可以**constexpr**：

```
class Point {
public:
    ...
    constexpr void setX(double newX) noexcept   // C++14
    { x = newX; }
    constexpr void setY(double newY) noexcept   // C++14
    { y = newY; }
    ...
};
```

这使得写这奇葩的函数成为可能：

```
// 返回p的映像（C++14）
constexpr Point reflection(const Point &p) noexcept
{
    Point result;     // create non-const Point

    result.setX(-p.xValue());
    result.setY(-p.yValue());

    return result;
}
```

用户的代码可能是这样的：

```
constexpr Point p1(9.4, 27.7);
constexpr Point p2(28.8, 5.3);
constexpr auto mid = midpoint(p1, p2);

constexpr auto reflectedMid =      // reflectedMid的值是（-19.1, -16.5）
    reflection(mid);                 //  而且在编译期间就知道了
```

本条款的建议是尽可能使用**constexpr**，然后现在我希望你能很清楚为什么：**constexpr**对象和**constexpr**函数比起non-constexpr对象和函数具有更广泛的语境。通过尽可能地使用**constexpr**，你最大化了对象和函数的可能使用的情况。

注意到**constexpr**是一个对象或函数接口的一部分是很重要的。**constexpr**表明“我可以用于需求常量表达式的上下文”，如果你把对象或者函数声明为**constexpr**，用户就有可能把它用于这种上下文。后来，如果你觉得你使用**constexpr**是个错误，然后你删除了它，这样就可能造成用户大量代码无法编译（为了调试添加I/O函数会导致这种问题，因为I/O语句通常不允许出现在**constexpr**函数），“尽可能使用**constexpr**”中的“尽可能”是你愿意作出长期的承诺，强行约束着**constexpr**的对象和函数（这句话太难了，我不知道我的理解有没问题：Part of “whenever possible”in “Use constexpr whenever possible” is your willingness to make a long-term commitment to the constraints it imposes on the objects and functions you apply it to.）。

总结

需要记住4点：

- **constexpr**对象是**const**的，它需用编译期间已知的值初始化。
- **constexpr**函数在传入编译期已知值作为参数时，会在编译期间生成结果。
- **constexpr**对象和函数比起non-constexpr对象和函数具有更广泛的语境。
- **constexpr**是对象和函数接口的一部分。