# Item 15:尽可能的使用constexpr

条款 15:尽可能的使用constexpr

如果要给C++11颁一个“最令人困惑新词”奖，**constexpr**十有八九会折桂。当用于对象上面，它本质上就是**const**的加强形式，但是当它用于函数上，意思就大不相同了。有必要消除困惑，因为你绝对会用它的，特别是当你发现**constexpr** “正合吾意”的时候。

从概念上来说，**constexpr**表明一个值不仅仅是常量，还是编译期可知的。这个表述并不全面，因为当**constexpr**被用于函数的时候，事情就有一些细微差别了。
为了避免我毁了结局带来的surprise，我现在只想说，你不能假设**constexpr**函数是**const**，也不能保证它们的（译注：返回）值是在编译期可知的。最有意思的是，这些是特性。关于**constexpr**函数返回的结果不需要是**const**，也不需要编译期可知这一点是良好的行为。

不过我们还是先从**constexpr**对象开始说起。这些对象，实际上，和**const**一样，它们是编译期可知的。（技术上来讲，它们的值在翻译期（translation）决议，所谓翻译不仅仅包含是编译（compilation）也包含链接（linking），除非你准备写C++的编译器和链接器，否则这些对你不会造成影响，所以你编程时无需担心，把这些**constexpr**对象值看做编译期决议也无妨的。）

编译期可知的值“享有特权”，它们可能被存放到只读存储空间中。对于那些嵌入式系统的开发者，这个特性是相当重要的。更广泛的应用是“其值编译期可知”的常量整数会出现在需要“整型常量表达式（**integral constant expression**）的`context`中，这类`context`包括数组大小，整数模板参数（包括`std::array`对象的长度），枚举量，对齐修饰符（译注：`alignas(val)`），等等。如果你想在这些`context`中使用变量，你一定会希望将它们声明为**constexpr**，因为编译器会确保它们是编译期可知的：

```
int sz;                                // 非constexpr变量
…
constexpr auto arraySize1 = sz;        // 错误！ sz的值在
                                       // 编译期不可知
std::array<int, sz> data1;             // 错误!一样的问题
constexpr auto arraySize2 = 10;        // 没问题，10是编译
                                       // 期可知常量
std::array<int, arraySize2> data2;     // 没问题，arraySize2是constexpr
```

注意**const**不提供**constexpr**所能保证之事，因为**const**对象不需要在编译期初始化它的值。

```
int sz;                                // 和之前一样
const auto arraySize = sz;             // 没问题，arraySize是sz的常量复制
std::array<int, arraySize> data;       // 错误，arraySize值在编译期不可知
```

简而言之，所有**constexpr**对象都是**const**，但不是所有**const**对象都是**constexpr**。如果你想编译器保证一个变量有一个可以放到那些需要编译期常量的上下文的值，你需要的工具是**constexpr**而不是**const**。

如果使用场景涉及函数，那 **constexpr**就更有趣了。如果实参是编译期常量，它们将产出编译期值；如果是运行时值，它们就将产出运行时值。这听起来就像你不知道它们要做什么一样，那么想是错误的，请这么看：

- **constexpr**函数可以用于需求编译期常量的上下文。如果你传给**constexpr**函数的实参在编译期可知，那么结果将在编译期计算。如果实参的值在编译期不知道，你的代码就会被拒绝。
- 当一个**constexpr**函数被一个或者多个编译期不可知值调用时，它就像普通函数一样，运行时计算它的结果。这意味着你不需要两个函数，一个用于编译期计算，一个用于运行时计算。**constexpr**

全做了。

假设我们需要一个数据结构来存储一个实验的结果，而这个实验可能以各种方式进行。实验期间风扇转速，温度等等都可能导致亮度值改变，亮度值可以是高，低，或者无。如果有n个实验相关的环境条件。它们每一个都有三个状态，最终可以得到的组合$$3^n$$个。储存所有实验结果的所有组合需要这个数据结构足够大。假设每个结果都是**int**并且**n**是编译期已知的（或者可以被计算出的），一个`std::array`是一个合理的选择。我们需要一个方法在编译期计算`3^n`。C++标准库提供了`std::pow`，它的数学意义正是我们所需要的，但是，对我们来说，这里还有两个问题。第一，`std::pow`是为浮点类型设计的 我们需要整型结果。第二，`std::pow`不是**constexpr**（即，使用编译期可知值调用得到的可能不是编译期可知的结果），所以我们不能用它作为`std::array`的大小。

幸运的是，我们可以应需写个`pow`。我将展示怎么快速完成它，不过现在让我们先看看它应该怎么被声明和使用：

```
constexpr                                   // pow是constexpr函数
int pow(int base, int exp) noexcept         // 绝不抛异常
{
 …                                          // 实现在这里
}
constexpr auto numConds = 5;                //条件个数
std::array<int, pow(3, numConds)> results; // 结果有3^numConds个元素
```

回忆下`pow`前面的**constexpr**没有告诉我们`pow`返回一个**const**值，它只说了如果**base**和**exp**是编译期常量，`pow`返回值可能是编译期常量。如果**base** 和/或 **exp**不是编译期常量，`pow`结果将会在运行时计算。这意味着**pow**不知可以用于像`std::array`的大小这种需要编译期常量的地方，它也可以用于运行时环境：

```
auto base = readFromDB("base");     // 运行时获取三个值
auto exp = readFromDB("exponent");
auto baseToExp = pow(base, exp);    // 运行时调用pow
```

因为**constexpr**函数必须能在编译期值调用的时候返回编译器结果，就必须对它的实现施加一些限制。这些限制在C++11和C++14标准间有所出入。

C++11中，**constexpr**函数的代码不超过一行语句：一个**return**。听起来很受限，但实际上有两个技巧可以扩展**constexpr**函数的表达能力。第一，使用三元运算符"?:"来代替**if-else**语句，第二，使用递归代替循环。因此**pow**可以像这样实现：

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

这样没问题，但是很难想象除了使用函数式语言的程序员外会觉得这样硬核的编程方式更好。在C++14中，**constexpr**函数的限制变得非常宽松了，所以下面的函数实现成为了可能；

```
constexpr int pow(int base, int exp) noexcept  // C++14
{
  auto result = 1;
  for (int i = 0; i < exp; ++i) result *= base;
  return result;
}
```

**constexpr**函数限制为只能获取和返回字面值类型，这基本上意味着具有那些类型的值能在编译期决定。在C++11中，除了void外的所有内置类型外还包括一些用户定义的字面值，因为构造函数和其他成员函数可以是**constexpr**：

```
class Point {
  public:
    constexpr Point(double xVal = 0, double yVal = 0) noexcept : x(xVal), y(yVal)
    {}
    constexpr double xValue() const noexcept { return x; }
    constexpr double yValue() const noexcept { return y; }

    void setX(double newX) noexcept { x = newX; }
    void setY(double newY) noexcept { y = newY; }
  private:
    double x, y;
};
```

**Point**的构造函数被声明为**constexpr**，因为如果传入的参数在编译期可知，**Point**的数据成员也能在编译器可知。因此**Point**就能被初始化为**constexpr**：

```
constexpr Point p1(9.4, 27.7); // 没问题，构造函数会在编译期“运行”
constexpr Point p2(28.8, 5.3); // 也没问题
```

类似的，xValue和yValue的getter函数也能是**constexpr**，因为如果对一个编译期已知的**Point**对象调用getter，数据成员x和y的值也能在编译期知道。这使得我们可以写一个**constexpr**函数里面调用**Point**的getter并初始化**constexpr**的对象：

```
constexpr
Point midpoint(const Point& p1, const Point& p2) noexcept
{
  return { (p1.xValue() + p2.xValue()) / 2,
           (p1.yValue() + p2.yValue()) / 2 };
}
constexpr auto mid = midpoint(p1, p2);
```

这太令人激动了。它意味着mid对象通过调用构造函数，getter和成员函数就能在只读内存中创建！它也意味着你可以在模板或者需要枚举量的表达式里面使用像`mid.xValue()*10`的表达式！它也意味着以前相对严格的某一行代码只能用于编译期，某一行代码只能用于运行时的界限变得模糊，一些运行时的普通计算能并入编译时。越多这样的代码并入，你的程序就越快。（当然，编译会花费更长时间）

在C++11中，有两个限制使得**Point**的成员函数`setX`和`setY`不能声明为**constexpr**。第一，它们修改它们操作的对象的状态，并且在C++11中，**constexpr**成员函数是隐式的const。第二，它们只能有void返回类型，void类型不是C++11中的字面值类型。这两个限制在C++14中放开了，所以C++14中Point的setter也能声明为**constexpr**：

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

现在也能写这样的函数：

```
constexpr Point reflection(const Point& p) noexcept
{
    Point result;
    result.setX(-p.xValue());
    result.setY(-p.yValue());
    return result;
}
```

客户端代码可以这样写：

```
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
     reflection(mid);                  // 在编译期可知
```

本章的建议是尽可能的使用**constexpr**，现在我希望大家已经明白缘由：**constexopr**对象和**constexpr**函数可以用于很多非**constexpr**不能使用的场景。使用**constexpr**关键字可以最大化你的对象和函数可以使用的场景。

还有个重要的需要注意的是**constexpr**是对象和函数接口的一部分。加上**constexpr**相当于宣称“我能在C++要求常量表达式的地方使用它”。如果你声明一个对象或者函数是constexpr，客户端程序员就会在那些场景中使用它。如果你后面认为使用constexpr是一个错误并想移除它，你可能造成大量客户端代码不能编译。**尽可能**的使用**constexpr**表示你需要长期坚持对某个对象或者函数施加这种限制。

记住

- **constexpr**对象是**cosnt**，它的值在编译期可知
- 当传递编译期可知的值时，**cosntexpr**函数可以产出编译期可知的结果