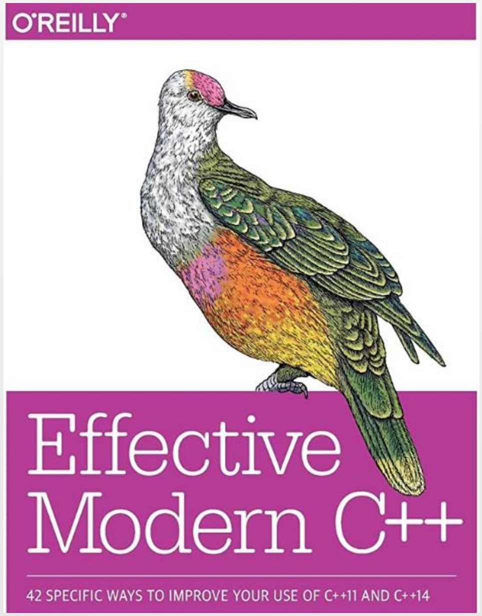
ITEM 15:
Use **constexpr** whenever possible.

Scott Meyers - Effective Modern C++



Presented By
@7eRoM

- constexpr indicates a value that's not only constant, it's known during compilation.
- They may be placed in read-only memory.
- Integral values that are constant and known during compilation can be used in contexts where C++ requires an integral constant expression.
- Such contexts include specification of array sizes, integral template arguments (including lengths of std::array objects), enumerator values, alignment specifiers, and more.
- All constexpr objects are const, but not all const objects are constexpr.

```cpp
int sz;                             // non-constexpr variable

…

constexpr auto arraySize1 = sz;     // error! sz's value not
                                    // known at compilation

std::array<int, sz> data1;          // error! same problem

constexpr auto arraySize2 = 10;     // fine, 10 is a

                                    // compile-time constant

std::array<int, arraySize2> data2;  // fine, arraySize2
                                    // is constexpr
```

```cpp
int sz;                             // as before

…

const auto arraySize = sz;          // fine, arraySize is
                                    // const copy of sz

std::array<int, arraySize> data;    // error! arraySize's value
                                    // not known at compilation
```

- When a constexpr function is called with one or more values that are not known during compilation, it acts like a normal function, computing its result at runtime. This means you don't need two functions to perform the same operation,

- Example: we need a data structure to hold the results of an experiment that can be run in a variety of ways
  - The lighting level can be high, low, or off
  - If there are n environmental conditions relevant to the experiment, each of which has three possible states, the number of combinations is 3^n
  - Storing experimental results for all combinations of conditions thus requires a data structure with enough room for 3^n values
  - we'd need a way to compute 3^n during compilation
  - std::pow is the mathematical functionality we need, but, there are two problems:
    - std::pow works on floating-point types, and we need an integral result.
    - std::pow isn't constexpr

```cpp
constexpr                                     // pow's a constexpr func
int pow(int base, int exp) noexcept           // that never throws
{
  …                                           // impl is below
}

constexpr auto numConds = 5;                        // # of conditions

std::array<int, pow(3, numConds)> results;          // results has
                                                    // 3^numConds
                                                    // elements
```

- If base and/or exp are not compile-time constants, pow's result will be computed at runtime.

```cpp
auto base = readFromDB("base");           // get these values
auto exp = readFromDB("exponent");        // at runtime

auto baseToExp = pow(base, exp);          // call pow function
                                          // at runtime
```

- In C++11, constexpr functions may contain no more than a single executable statement: a return

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

- In C++14, the restrictions on constexpr functions are substantially looser

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

  return result;
}
```

- constexpr functions are limited to taking and returning literal types. It means the types that can have values determined during compilation
- In C++11
  - all built-in types except void qualify
  - user-defined types may be literal, because constructors and other member functions may be 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;
};
```

- Points so initialized could thus be constexpr

```
constexpr Point p1(9.4, 27.7);      // fine, "runs" constexpr
                                    // ctor during compilation

constexpr Point p2(28.8, 5.3);      // also fine
```

- The getters xValue and yValue can be constexpr
- It is possible to write constexpr functions that call Point's getters and to initialize constexpr objects with the results of such functions:

```
constexpr
Point midpoint(const Point& p1, const Point& p2) noexcept
{
  return { (p1.xValue() + p2.xValue()) / 2,    // call constexpr
           (p1.yValue() + p2.yValue()) / 2 };  // member funcs
}

constexpr auto mid = midpoint(p1, p2);     // init constexpr
                                           // object w/result of
                                           // constexpr function
```

- In C++11, two restrictions prevent Point's member functions setX and setY from being declared constexpr:
    - they modify the object they operate on, and in C++11, constexpr member functions are implicitly const
    - they have void return types, and void isn't a literal type in C++11.
- Both these restrictions are lifted in C++14

```
class Point {
public:
  …

  constexpr void setX(double newX) noexcept     // C++14
  { x = newX; }

  constexpr void setY(double newY) noexcept     // C++14
  { y = newY; }

  …

};
```

- That makes it possible to write functions like this:

```
// return reflection of p with respect to the origin (C++14)
constexpr Point reflection(const Point& p) noexcept
{
  Point result;                       // create non-const Point

  result.setX(-p.xValue());           // set its x and y values
  result.setY(-p.yValue());

  return result;                      // return copy of it
}
```

- Client code could look like this:

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

constexpr auto reflectedMid =         // reflectedMid's value is
  reflection(mid);                    // (-19.1 -16.5) and known
                                      // during compilation
```