CppCoreGuidelines, part 2
Constants and Immutability

Juanmi Huertas
R&D Software Engineer @ HP

CppCoreGuideline index

P: Philosophy
I: Interfaces
F: Functions
C: Classes and class hierarchies
Enum: Enumerations
R: Resource management
ES: Expressions and statements
E: Error handling
Per: Performance

Con: Constants and immutability
T: Templates and generic programming
CP: Concurrency
SL: The Standard library
SF: Source files
CPL: C-style programming
Pro: Profiles
N: Non-Rules and myths
NL: Naming and layout

Previously... on CppCoreGuidelines, part 1...

1) Meaningful code contains it's own meaning.
P.1: Express ideas directly in code, P.3: Express intent.

2) Legible code is easy to read an to understand.
P.11: Encapsulate messy constructs, rather than spreading through the code.

3) Error-free code has no errors.
P.4: Ideally, a program should be statically type safe, P.5: Prefer compile-time checking to run-time checking,
P.6: What cannot be checked at compile time should becheckable at run time,
P.7: Catch run-time errors early.

4) 'Cheap' code perform faster.
P.8: Don't leak any resources, P.9: Don't waste time or space, P.10: Prefer immutable data to mutable data.

5) Standard code follows Software Engineer principles.
P.2: Write in ISO Standard C++, P.12: Use supporting tools as appropriate,
P.13: Use support libraries as appropriate.

CppCoreGuideline index

P: Philosophy
I: Interfaces
F: Functions
C: Classes and class hierarchies
Enum: Enumerations
R: Resource management
ES: Expressions and statements
E: Error handling
Per: Performance

Con: Constants and immutability
T: Templates and generic programming
CP: Concurrency
SL: The Standard library
SF: Source files
CPL: C-style programming
Pro: Profiles
N: Non-Rules and myths
NL: Naming and layout

1) Use `const` for your types

Express intent. Express ideas through code.

Help other people – and yourself – with `const`.

```
for (int i : c)          // bad
   cout << i << '\n';


for (const int i : c)    // good
    cout << i << '\n';
```

```
void f()
{
  int x = 7;
  const int y = 9;

  for (;;)
  {
    // ...
  }
  // ...
}
```

We assume that x is going to change.

We assume that y is not going to change.

Con.1: By default, make objects immutable
Con.4: Use `const` to define objects with values that do not change after construction

1.a) Use `const` for your types

NR.1: Don't: All declarations should be at the top of a function
ES.21: Don't introduce a variable (or constant) before you need to use it

```
int use(int x)
{
    int i = 0;
    char c = 'a';
    double d = 0.;

    // ... lots of stuff

    // now I use i, c and d
}
```

```
void f()
{
  // ...
  // ...
  {
     int x = 7;
     g(x);
  }
}
```

Con.1: By default, make objects immutable
Con.4: Use `const` to define objects with values that do not change after construction

1.b) Use const for your types

```
int main()
{
  std::string s;
  s = “A Somewhat Rather Long String”;
}

int main()
{
  const std::string s {“A Somewhat Rather Long String”};
}
```

Collateral benefits:

1) Performance.
   Don’t do more work than you have to.

2) Initialize everything.
   Easier to read code.
   Names declared in the precise scope.
   Reduces refactoring.

Con.1: By default, make objects immutable
Con.4: Use const to define objects with values that do not change after construction

1.c) Use const for your types

But what about complex initialization?

```
int main()
{
  const int i = std::rand();
  std::string s;
  switch(i%4)
  {
    case 0:
      s = “long string is mod 0”;
      break;
    case 1:
      s = “long string is mod 1”;
      break;
    case 2:
      s = “long string is mod 2”;
      break;
    case 3:
      s = “long string is mod 3”;
      break;
}
```

Use Lambdas! IIFE

```
int main()
{
  const int i = std::rand();
  const std::string s = [&](){
    switch(i%4)
    {
      case 0:
        return “long string is mod 0”;
      case 1:
        return “long string is mod 1”;
      case 2:
        return “long string is mod 2”;
      case 3:
        return “long string is mod 3”;
    }
  }();
}
```

Con.1: By default, make objects immutable
Con.4: Use const to define objects with values that do not change after construction

2) Use `const` for your member functions

Always express intent.
It will help you and fellow programmers.

The compiler is your friend.
Help him help you.

You will be able to detect issues.

```
class Point {
    int x, y;

public:
    int getx() const
    {
      return x;
    }
    // ...
};

void f(const Point& pt) {
    int x = pt.getx();
}
```

Con.2: By default, make member functions `const`

2.a) Use `const` for your member functions

```
class polygon {
public:
  polygon() : area{-1} {}
  //...
  double get_area() const {
    if( area < 0 )
       calc_area();
    return area;
  }

private:
  void calc_area() const {
     area = /* area calculation */;
  }

  vector<point> points;
  mutable atomic<double> area;
};
```

What does happen here?

How to do something non-const within a const method?

```
  void calc_area() const {
     area = /* area calculation */;
  }

  vector<point> points;
  mutable atomic<double> area;
};
```

Con.2: By default, make member functions `const`

2.b) Use `const` for your member functions

```
class Person {
public:
  const std::string& name_good() const;  // Right: the caller can't change the Person's name
  std::string& name_evil() const;        // Wrong: the caller can change the Person's name
  int age() const;                       // Also right: the caller can't change the Person's age
  // ...
};
void myCode(const Person& p)  // myCode() promises not to change the Person object...
{
  p.name_evil() = "Igor";     // But myCode() changed it anyway!!
}
```

```
class Fred { /*...*/ };
class MyFredList {
public:
  const Fred& operator[] (unsigned index) const;  // Subscript operators often come in pairs
  Fred&       operator[] (unsigned index);        // Subscript operators often come in pairs
  // ...
};
```

Con.2: By default, make member functions `const`

3) Use `const` for your parameters

Express intent and help the compiler help you.

```
void f(char *p);

void g(const char*p);
```

```
void f(const &i)
{
  i = 42; // ERROR
}
```

ES.50: Don't cast away `const`

```
void f(const &i)
{
  const_cast<int&>(i) = 42;
}
```

```
static int i = 0;
static const int j = 0;

f(i); // silent side effect
f(j); // undefined behavior
```

Con.3: By default, pass pointers and references to `const`s

3.a) Use `const` for your parameters

ES.50: Don't cast away `const`

```
class Bar;

class Foo {
public:
    // BAD, duplicates logic
    Bar& get_bar() {
        /* complex logic around getting a non-const reference to my_bar */
    }

    const Bar& get_bar() const {
        /* same complex logic around getting a const reference to my_bar */
    }
private:
    Bar my_bar;
};
```

Con.3: By default, pass pointers and references to `const`s

3.a) Use `const` for your parameters

ES.50: Don't cast away `const`

```
class Bar;

class Foo {
public:
    // not great, non-const calls const version but resorts to const_cast
    Bar& get_bar() {
        return const_cast<Bar&>(static_cast<const Foo&>(*this).get_bar());
    }

    const Bar& get_bar() const {
        /* the complex logic around getting a const reference to my_bar */
    }
private:
    Bar my_bar;
};
```

Con.3: By default, pass pointers and references to `const`s

3.a) Use `const` for your parameters

ES.50: Don't cast away `const`

```
class Bar;

class Foo {
public:                          // good
          Bar& get_bar()       { return get_bar_impl(*this); }
    const Bar& get_bar() const { return get_bar_impl(*this); }

private:
    Bar my_bar;

    template<class T>            // good, deduces whether T is const or non-const
    static auto get_bar_impl(T& t) -> decltype(t.get_bar())
        { /* the complex logic around getting a possibly-const reference to my_bar */ }
};
```

Con.3: By default, pass pointers and references to `const`s

4) Use `constexpr` as much as you can

Express intent and help the compiler help you.

```
double x = f(2);            // possible run-time evaluation
const double y = f(2);      // possible run-time evaluation
constexpr double z = f(2);  // error unless f(2) can be evaluated at compile time
```

STOP USING MACROS

```
#define VERSION_MAJOR 1
#define VERSION_MINOR 0
#define VERSION_RELEASE 0

constexpr unsigned int version_major = 1U;
constexpr unsigned int version_minor = 0U;
constexpr unsigned int version_release = 0U;
```

Video of constexpr ALL the things: Ben Deane & Jason Turner creating a JSON parser and a regex for compile time. Link.

Con.5: Use `constexpr` for values that can be computed at compiler time

4.a) Use `constexpr` as much as you can

Reduce the number of magic constants

Lambdas are constexpr by default (C++17)

if constexpr exists (C++17)

Jason Turner: compile time rand gen.

```
template<int N>
class list
{ };

constexpr int sqr1(int arg)
{ return arg * arg; }

int sqr2(int arg)
{ return arg * arg; }

int main()
{
  const int X = 2;
  list<sqr1(X)> mylist1;  // OK
  list<sqr2(X)> mylist2;  // wrong
  return 0;
}
```

Con.5: Use `constexpr` for values that can be computed at compiler time

Summing up (multimap<CppGuidelines::value_type, this_talk:: ::value_type>)

1) Use `const` for your types.

Con.1: By default, makes objects immutable,
Con.4: Use `const` define objects with values that do not change after construction.

2) Use `const` for your member functions

Con.2: By default, make member functions `const`.

3) Use `const` for your parameters.

Con.3: By default, pass pointers and referneces to `consts`.

4) Use `constexpr` as much as you can.

Con.5: Use `constexpr` for values that can be computed at compile time.

Next topic?

P: Philosophy 
I: Interfaces
F: Functions
C: Classes and class hierarchies
Enum: Enumerations
R: Resource management
ES: Expressions and statements
E: Error handling
Per: Performance

Con: Constants and immutability 
T: Templates and generic programming
CP: Concurrency
SL: The Standard library
SF: Source files
CPL: C-style programming
Pro: Profiles
N: Non-Rules and myths
NL: Naming and layout

References

CppCoreGuidelines (duh!): Link
Isocpp, FAQ: const-correctness: Link
Jason Turner – C++ Weekly: Link
Richard Powell – The importance of being Const. cppcon'15: Link
Jason Turner – Practical Performance Tips. cppcon'16: Link
Ben Dean & Jason Turner – Constexpr all the things. cppcon'17: Link