The Importance of Being `const`

Richard Powell <rmpowell@me.com>
9/23/2015 - v1.7

The Importance of Being `const`

- “[`const`] allows you to communicate to both compilers and other programmers that a value should remain invariant. Whenever that is true, you should be sure to say so, because that way you enlist your compilers’ aid in making sure the constraint isn’t violated.”

- Scott Meyers, Effective C++ 3rd Edition, Item 3

The Importance of Being `const`

- `const` variables, data members, methods and arguments add a level of compile-time type checking; it is better to detect errors as soon as possible. Therefore we strongly recommend that you use `const` whenever it makes sense to do so.

http://google-styleguide.googlecode.com/svn/trunk/cppguide.html#Use_of_const

The Importance of Being `const`

- “A **`const`** object is an object of type **`const`** T or a **non-mutable** subobject of such an object.” - ISO N3690: § 3.9.3 CV-qualifiers [basic.type.qualifier]

```
const T t; // t is a constant T
T const t; // same thing
```

- **mutate** |'myo͞oˌtāt| : (verb) change or cause to change in form or nature

- Declaring an object **`const`** means expressions which modify the object are not allowed.

```
int main()
{
  int a = 0;
  long const b = 1;

  ++a;                                    // 1

  b++;                                    // 2

  if (b > 4)                              // 3
    printf("b is greater than 4\n");

  if (b == 4)                             // 4
    printf("b is equal to 4\n");

  if (b >>= a)                            // 5
    printf("greater than or equal\n");

  a = 0 ? ++b : b;                        // 6
}
```

Valid Expression

Line 1 ☐

Line 2 ☐

Line 3 ☐

Line 4 ☐

Line 5 ☐

Line 6 ☐

```
int main()
{
  int a = 0;
  long const b = 1;

  ++a;                                  // 1

  b++;                                  // 2

  if (b > 4)                            // 3
    printf("b is greater than 4\n");

  if (b == 4)                           // 4
    printf("b is equal to 4\n");

  if (b >>= a)                          // 5
    printf("greater than or equal\n");

  a = 0 ? ++b : b;                      // 6
}
```

Valid Expression

Line 1 ☑

Line 2 ☐

Line 3 ☐

Line 4 ☐

Line 5 ☐

Line 6 ☐

```
int main()
{
  int a = 0;
  long const b = 1;

  ++a;                                    // 1

  b++;                                    // 2

  if (b > 4)                              // 3
    printf("b is greater than 4\n");

  if (b == 4)                             // 4
    printf("b is equal to 4\n");

  if (b >>= a)                            // 5
    printf("greater than or equal\n");

  a = 0 ? ++b : b;                        // 6
}
```

Valid Expression

Line 1 ☑

Line 2 ☒

Line 3 ☐

Line 4 ☐

Line 5 ☐

Line 6 ☐

```
int main()
{
  int a = 0;
  long const b = 1;

  ++a;                                    // 1

  b++;                                    // 2

  if (b > 4)                              // 3
    printf("b is greater than 4\n");

  if (b == 4)                             // 4
    printf("b is equal to 4\n");

  if (b >>= a)                            // 5
    printf("greater than or equal\n");

  a = 0 ? ++b : b;                        // 6
}
```

Valid Expression

Line 1 ☑

Line 2 ☒

Line 3 ☑

Line 4 ☐

Line 5 ☐

Line 6 ☐

```
int main()
{
  int a = 0;
  long const b = 1;

  ++a;                                    // 1

  b++;                                    // 2

  if (b > 4)                              // 3
    printf("b is greater than 4\n");

  if (b == 4)                             // 4
    printf("b is equal to 4\n");

  if (b >>= a)                            // 5
    printf("greater than or equal\n");

  a = 0 ? ++b : b;                        // 6
}
```

Valid Expression

Line 1 ☑

Line 2 ☒

Line 3 ☑

Line 4 ☑

Line 5 ☐

Line 6 ☐

```
int main()
{
  int a = 0;
  long const b = 1;

  ++a;                                  // 1

  b++;                                  // 2

  if (b > 4)                            // 3
    printf("b is greater than 4\n");

  if (b == 4)                           // 4
    printf("b is equal to 4\n");

  if (b >>= a)                          // 5
    printf("greater than or equal\n");

  a = 0 ? ++b : b;                      // 6
}
```

Valid Expression

Line 1 ☑

Line 2 ☒

Line 3 ☑

Line 4 ☑

Line 5 ☒

Line 6 ☐

```
int main()
{
  int a = 0;
  long const b = 1;

  ++a;                                  // 1

  b++;                                  // 2

  if (b > 4)                            // 3
    printf("b is greater than 4\n");

  if (b == 4)                           // 4
    printf("b is equal to 4\n");

  if (b >>= a)                          // 5
    printf("greater than or equal\n");

  a = 0 ? ++b : b;                      // 6
}
```

Valid Expression

Line 1 ☑

Line 2 ☒

Line 3 ☑

Line 4 ☑

Line 5 ☒

Line 6 ☒

```
int a[] = { 1, 2, 3 };

int main()
{
  int * b = a;
  b++;                    // 1
  *b = 20;                // 2

  int const * c = a;
  c++;                    // 3
  *c = 20;                // 4

  int * const d = a;
  d++;                    // 5
  *d = 20;                // 6

  int const * const e = a;
  e++;                    // 7
  *e = 20;                // 8
}
```

	Valid Expression
Line 1	☐
Line 2	☐
Line 3	☐
Line 4	☐
Line 5	☐
Line 6	☐
Line 7	☐
Line 8	☐

```
int a[] = { 1, 2, 3 };

int main()
{
  int * b = a;
  b++;                    // 1
  *b = 20;                // 2

  int const * c = a;
  c++;                    // 3
  *c = 20;                // 4

  int * const d = a;
  d++;                    // 5
  *d = 20;                // 6

  int const * const e = a;
  e++;                    // 7
  *e = 20;                // 8
}
```

	Valid Expression
Line 1	☑
Line 2	☐
Line 3	☐
Line 4	☐
Line 5	☐
Line 6	☐
Line 7	☐
Line 8	☐

```
int a[] = { 1, 2, 3 };

int main()
{
  int * b = a;
  b++;                    // 1
  *b = 20;                // 2

  int const * c = a;
  c++;                    // 3
  *c = 20;                // 4

  int * const d = a;
  d++;                    // 5
  *d = 20;                // 6

  int const * const e = a;
  e++;                    // 7
  *e = 20;                // 8
}
```

	Valid Expression
Line 1	☑
Line 2	☑
Line 3	☐
Line 4	☐
Line 5	☐
Line 6	☐
Line 7	☐
Line 8	☐

```
int a[] = { 1, 2, 3 };

int main()
{
  int * b = a;
  b++;                    // 1
  *b = 20;                // 2

  int const * c = a;
  c++;                    // 3
  *c = 20;                // 4

  int * const d = a;
  d++;                    // 5
  *d = 20;                // 6

  int const * const e = a;
  e++;                    // 7
  *e = 20;                // 8
}
```

	Valid Expression
Line 1	☑
Line 2	☑
Line 3	☑
Line 4	☐
Line 5	☐
Line 6	☐
Line 7	☐
Line 8	☐

```
int a[] = { 1, 2, 3 };

int main()
{
  int * b = a;
  b++;                    // 1
  *b = 20;                // 2

  int const * c = a;
  c++;                    // 3
  *c = 20;                // 4

  int * const d = a;
  d++;                    // 5
  *d = 20;                // 6

  int const * const e = a;
  e++;                    // 7
  *e = 20;                // 8
}
```

	Valid Expression
Line 1	☑
Line 2	☑
Line 3	☑
Line 4	☒
Line 5	☐
Line 6	☐
Line 7	☐
Line 8	☐

```
int a[] = { 1, 2, 3 };

int main()
{
  int * b = a;
  b++;                    // 1
  *b = 20;                // 2

  int const * c = a;
  c++;                    // 3
  *c = 20;                // 4

  int * const d = a;
  d++;                    // 5
  *d = 20;                // 6

  int const * const e = a;
  e++;                    // 7
  *e = 20;                // 8
}
```

	Valid Expression
Line 1	☑
Line 2	☑
Line 3	☑
Line 4	☒
Line 5	☒
Line 6	☐
Line 7	☐
Line 8	☐

```
int a[] = { 1, 2, 3 };

int main()
{
  int * b = a;
  b++;                    // 1
  *b = 20;                // 2

  int const * c = a;
  c++;                    // 3
  *c = 20;                // 4

  int * const d = a;
  d++;                    // 5
  *d = 20;                // 6

  int const * const e = a;
  e++;                    // 7
  *e = 20;                // 8
}
```

	Valid Expression
Line 1	☑
Line 2	☑
Line 3	☑
Line 4	☒
Line 5	☒
Line 6	☑
Line 7	☐
Line 8	☐

```
int a[] = { 1, 2, 3 };

int main()
{
  int * b = a;
  b++;                    // 1
  *b = 20;                // 2

  int const * c = a;
  c++;                    // 3
  *c = 20;                // 4

  int * const d = a;
  d++;                    // 5
  *d = 20;                // 6

  int const * const e = a;
  e++;                    // 7
  *e = 20;                // 8
}
```

	Valid Expression
Line 1	☑
Line 2	☑
Line 3	☑
Line 4	☒
Line 5	☒
Line 6	☑
Line 7	☒
Line 8	☐

```
int a[] = { 1, 2, 3 };

int main()
{
  int * b = a;
  b++;                    // 1
  *b = 20;                // 2

  int const * c = a;
  c++;                    // 3
  *c = 20;                // 4

  int * const d = a;
  d++;                    // 5
  *d = 20;                // 6

  int const * const e = a;
  e++;                    // 7
  *e = 20;                // 8
}
```

Valid Expression

Line 1 ☑

Line 2 ☑

Line 3 ☑

Line 4 ☒

Line 5 ☒

Line 6 ☑

Line 7 ☒

Line 8 ☒

The Importance of Being `const`

- The secret is to read from Right to Left
- (Also why the form `T const` is becoming more popular)

```
const int * p1;          // p1 is a pointer to "int const"/constant ints

int const * p2;          // p2 is a pointer to "const"ant ints

int * const p3;          // p3 is a "const"ant pointer ints

int const * const p4; // p4 is a "const"ant pointer to "const"ant ints
```

The Importance of Being `const`

- “A **`const`** object is an object of type **`const`** T or a **non-mutable** subobject of such an object.” - ISO N3690: § 3.9.3 CV-qualifiers [basic.type.qualifier]

```
T t;
```

```
T const t;
```

The Importance of Being `const`

- “A **`const`** object is an object of type **`const`** T or a **non-mutable** subobject of such an object.” - ISO N3690: § 3.9.3 CV-qualifiers [basic.type.qualifier]

```
T t;
```


```
T const t;
```

The Importance of Being `const`

- “A **`const`** object is an object of type **`const`** T or a **non-mutable** subobject of such an object.” - ISO N3690: § 3.9.3 CV-qualifiers [basic.type.qualifier]

```
T t;
```


```
T const t;
```

“Once you’re const, you are ensconced”
- Susan (my lovely wife)

The Importance of Being `const`

- Well, not entirely…

```
T t;
```


```
T const t;
```

The Importance of Being `const`

- Well, not entirely…

The Importance of Being `const`

- Well, not entirely…

The Importance of Being `const`

- Well, not entirely…

```
T t;
```


```
T const t;
```

```
const_cast<T>()
```

The Importance of Being `const`

- Well, not entirely…

```
T t;
```


```
T const t;
```

The Importance of Being `const`

- The compiler is free to add "**`const`**"ness (implicitly promote) an object to satisfy the expression, but it cannot take it away.

```
int d = 1;

int * p1 = &d;        // p1 points to (mutable) int

int const * p2 = p1; // p2 points to "const"ant int, const added

int * p3 = p2;        // ERROR! Cannot remove "const"ness of p2

int * const p4 = p1; // p4 is a "const"ant pointer to int, const added to ptr


int const d2 = 2;

int * p5 = &d2;       // ERROR! Cannot remove "const"ness of d2
```

The Importance of Being `const`

- What holds for pointers holds for references (almost).

```
int d = 1;

int & r1 = d;          // r1 refers to (mutable) int

int const & r2 = r1; // r2 refers to "const"ant int, const added

int & r3 = r2;         // ERROR! Cannot remove "const"ness of r2

int & const r4 = r1; // ERROR! "const" qualifiers may not be added to ref


int const d2 = 2;

int & r5 = d2;         // ERROR! Cannot remove "const"ness of d2
```

```
struct Foo
{
  int a = 0;
  int const b = 1;
};

int main()
{
  Foo f{};

  f.a++;                          // 1
  f.b++;                          // 2

  Foo const cf{};

  cf.a++;                         // 3
  cf.b++;                         // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  ptr_f->a++;                     // 5
  const_ptr_f->a++;               // 6

  ptr_f++;                        // 7
  const_ptr_f++;                  // 8
}
```

Valid Expression

Line 1 ☐

Line 2 ☐

Line 3 ☐

Line 4 ☐

Line 5 ☐

Line 6 ☐

Line 7 ☐

Line 8 ☐

```
struct Foo
{
  int a = 0;
  int const b = 1;
};

int main()
{
  Foo f{};

  f.a++;                          // 1
  f.b++;                          // 2

  Foo const cf{};

  cf.a++;                         // 3
  cf.b++;                         // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  ptr_f->a++;                     // 5
  const_ptr_f->a++;               // 6

  ptr_f++;                        // 7
  const_ptr_f++;                  // 8
}
```

	Valid Expression
Line 1	☑
Line 2	☐
Line 3	☐
Line 4	☐
Line 5	☐
Line 6	☐
Line 7	☐
Line 8	☐

```
struct Foo
{
  int a = 0;
  int const b = 1;
};

int main()
{
  Foo f{};

  f.a++;                          // 1
  f.b++;                          // 2

  Foo const cf{};

  cf.a++;                         // 3
  cf.b++;                         // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  ptr_f->a++;                     // 5
  const_ptr_f->a++;               // 6

  ptr_f++;                        // 7
  const_ptr_f++;                  // 8
}
```

Valid Expression

Line 1 

Line 2 

Line 3 ☐

Line 4 ☐

Line 5 ☐

Line 6 ☐

Line 7 ☐

Line 8 ☐

```
struct Foo
{
  int a = 0;
  int const b = 1;
};

int main()
{
  Foo f{};

  f.a++;                          // 1
  f.b++;                          // 2

  Foo const cf{};

  cf.a++;                         // 3
  cf.b++;                         // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  ptr_f->a++;                     // 5
  const_ptr_f->a++;               // 6

  ptr_f++;                        // 7
  const_ptr_f++;                  // 8
}
```

Valid Expression

Line 1 

Line 2 

Line 3 

Line 4 ☐

Line 5 ☐

Line 6 ☐

Line 7 ☐

Line 8 ☐

```
struct Foo
{
  int a = 0;
  int const b = 1;
};

int main()
{
  Foo f{};

  f.a++;                          // 1
  f.b++;                          // 2

  Foo const cf{};

  cf.a++;                         // 3
  cf.b++;                         // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  ptr_f->a++;                     // 5
  const_ptr_f->a++;               // 6

  ptr_f++;                        // 7
  const_ptr_f++;                  // 8
}
```

Valid Expression

Line 1 

Line 2 

Line 3 

Line 4 

Line 5 ☐

Line 6 ☐

Line 7 ☐

Line 8 ☐

```
struct Foo
{
  int a = 0;
  int const b = 1;
};

int main()
{
  Foo f{};

  f.a++;                          // 1
  f.b++;                          // 2

  Foo const cf{};

  cf.a++;                         // 3
  cf.b++;                         // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  ptr_f->a++;                     // 5
  const_ptr_f->a++;               // 6

  ptr_f++;                        // 7
  const_ptr_f++;                  // 8
}
```

Valid Expression

Line 1 

Line 2 

Line 3 

Line 4 

Line 5 

Line 6 ☐

Line 7 ☐

Line 8 ☐

```
struct Foo
{
  int a = 0;
  int const b = 1;
};

int main()
{
  Foo f{};

  f.a++;                          // 1
  f.b++;                          // 2

  Foo const cf{};

  cf.a++;                         // 3
  cf.b++;                         // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  ptr_f->a++;                     // 5
  const_ptr_f->a++;               // 6

  ptr_f++;                        // 7
  const_ptr_f++;                  // 8
}
```

Valid Expression

Line 1 

Line 2 

Line 3 

Line 4 

Line 5 

Line 6 

Line 7 ☐

Line 8 ☐

```
struct Foo
{
  int a = 0;
  int const b = 1;
};

int main()
{
  Foo f{};

  f.a++;                          // 1
  f.b++;                          // 2

  Foo const cf{};

  cf.a++;                         // 3
  cf.b++;                         // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  ptr_f->a++;                     // 5
  const_ptr_f->a++;               // 6

  ptr_f++;                        // 7
  const_ptr_f++;                  // 8
}
```

Valid Expression

Line 1 

Line 2 

Line 3 

Line 4 

Line 5 

Line 6 

Line 7 

Line 8 

```
struct Foo
{
  int a = 0;
  int const b = 1;
};

int main()
{
  Foo f{};

  f.a++;                          // 1
  f.b++;                          // 2

  Foo const cf{};

  cf.a++;                         // 3
  cf.b++;                         // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  ptr_f->a++;                     // 5
  const_ptr_f->a++;               // 6

  ptr_f++;                        // 7
  const_ptr_f++;                  // 8
}
```

Valid Expression

Line 1 

Line 2 

Line 3 

Line 4 

Line 5 

Line 6 

Line 7 

Line 8 

```
int data;

struct Foo
{
   int * a = &data;
   int const * b = &data;
};

int main()
{
  Foo f{};

  *f.a = 20;                    // 1
  *f.b = 20;                    // 2

  Foo const cf{};

  *cf.a = 20;                   // 3
  *cf.b = 20;                   // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  *ptr_f->a = 20;               // 5
  *ptr_f->b = 20;               // 6

  *const_ptr_f->a = 20;         // 7
  *const_ptr_f->b = 20;         // 8
}
```

Valid Expression

Line 1 ☐

Line 2 ☐

Line 3 ☐

Line 4 ☐

Line 5 ☐

Line 6 ☐

Line 7 ☐

Line 8 ☐

```
int data;

struct Foo
{
   int * a = &data;
   int const * b = &data;
};

int main()
{
  Foo f{};

  *f.a = 20;                    // 1
  *f.b = 20;                    // 2

  Foo const cf{};

  *cf.a = 20;                   // 3
  *cf.b = 20;                   // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  *ptr_f->a = 20;               // 5
  *ptr_f->b = 20;               // 6

  *const_ptr_f->a = 20;         // 7
  *const_ptr_f->b = 20;         // 8
}
```

	Valid Expression
Line 1	☑
Line 2	☐
Line 3	☐
Line 4	☐
Line 5	☐
Line 6	☐
Line 7	☐
Line 8	☐

```
int data;

struct Foo
{
   int * a = &data;
   int const * b = &data;
};

int main()
{
  Foo f{};

  *f.a = 20;                    // 1
  *f.b = 20;                    // 2

  Foo const cf{};

  *cf.a = 20;                   // 3
  *cf.b = 20;                   // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  *ptr_f->a = 20;               // 5
  *ptr_f->b = 20;               // 6

  *const_ptr_f->a = 20;         // 7
  *const_ptr_f->b = 20;         // 8
}
```

Valid Expression

Line 1 ☑

Line 2 ☒

Line 3 ☐

Line 4 ☐

Line 5 ☐

Line 6 ☐

Line 7 ☐

Line 8 ☐

```
int data;

struct Foo
{
   int * a = &data;
   int const * b = &data;
};

int main()
{
  Foo f{};

  *f.a = 20;                    // 1
  *f.b = 20;                    // 2

  Foo const cf{};

  *cf.a = 20;                   // 3
  *cf.b = 20;                   // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  *ptr_f->a = 20;               // 5
  *ptr_f->b = 20;               // 6

  *const_ptr_f->a = 20;         // 7
  *const_ptr_f->b = 20;         // 8
}
```

Valid Expression

Line 1 ☑

Line 2 ☒

Line 3 ☑

Line 4 ☐

Line 5 ☐

Line 6 ☐

Line 7 ☐

Line 8 ☐

```
int data;

struct Foo
{
   int * a = &data;
   int const * b = &data;
};

int main()
{
  Foo f{};

  *f.a = 20;                    // 1
  *f.b = 20;                    // 2

  Foo const cf{};

  *cf.a = 20;                   // 3
  *cf.b = 20;                   // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  *ptr_f->a = 20;               // 5
  *ptr_f->b = 20;               // 6

  *const_ptr_f->a = 20;         // 7
  *const_ptr_f->b = 20;         // 8
}
```

Valid Expression

Line 1 ☑

Line 2 ☒

Line 3 ☑

Line 4 ☒

Line 5 ☐

Line 6 ☐

Line 7 ☐

Line 8 ☐

```
int data;

struct Foo
{
   int * a = &data;
   int const * b = &data;
};

int main()
{
  Foo f{};

  *f.a = 20;                    // 1
  *f.b = 20;                    // 2

  Foo const cf{};

  *cf.a = 20;                   // 3
  *cf.b = 20;                   // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  *ptr_f->a = 20;               // 5
  *ptr_f->b = 20;               // 6

  *const_ptr_f->a = 20;         // 7
  *const_ptr_f->b = 20;         // 8
}
```

Valid Expression

Line 1 ☑

Line 2 ☒

Line 3 ☑

Line 4 ☒

Line 5 ☑

Line 6 ☐

Line 7 ☐

Line 8 ☐

```
int data;

struct Foo
{
   int * a = &data;
   int const * b = &data;
};

int main()
{
  Foo f{};

  *f.a = 20;                    // 1
  *f.b = 20;                    // 2

  Foo const cf{};

  *cf.a = 20;                   // 3
  *cf.b = 20;                   // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  *ptr_f->a = 20;               // 5
  *ptr_f->b = 20;               // 6

  *const_ptr_f->a = 20;         // 7
  *const_ptr_f->b = 20;         // 8
}
```

Valid Expression

Line 1 ☑

Line 2 ☒

Line 3 ☑

Line 4 ☒

Line 5 ☑

Line 6 ☒

Line 7 ☐

Line 8 ☐

```
int data;

struct Foo
{
   int * a = &data;
   int const * b = &data;
};

int main()
{
  Foo f{};

  *f.a = 20;                    // 1
  *f.b = 20;                    // 2

  Foo const cf{};

  *cf.a = 20;                   // 3
  *cf.b = 20;                   // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  *ptr_f->a = 20;               // 5
  *ptr_f->b = 20;               // 6

  *const_ptr_f->a = 20;         // 7
  *const_ptr_f->b = 20;         // 8
}
```

Valid Expression

Line 1 ☑

Line 2 ☒

Line 3 ☑

Line 4 ☒

Line 5 ☑

Line 6 ☒

Line 7 ☑

Line 8 ☐

```
int data;

struct Foo
{
   int * a = &data;
   int const * b = &data;
};

int main()
{
  Foo f{};

  *f.a = 20;                    // 1
  *f.b = 20;                    // 2

  Foo const cf{};

  *cf.a = 20;                   // 3
  *cf.b = 20;                   // 4

  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  *ptr_f->a = 20;               // 5
  *ptr_f->b = 20;               // 6

  *const_ptr_f->a = 20;         // 7
  *const_ptr_f->b = 20;         // 8
}
```

Valid Expression

Line 1 ☑

Line 2 ☒

Line 3 ☑

Line 4 ☒

Line 5 ☑

Line 6 ☒

Line 7 ☑

Line 8 ☒

```
struct Foo{};

void funct1(Foo *) { printf("funct1\n"); }
void funct2(Foo const *) { printf("funct2\n"); }
void funct3(Foo *) { printf("funct3(*)\n"); }
void funct3(Foo const *) { printf("funct3(const*)\n"); }

int main()
{
  Foo f{};
  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  funct1(ptr_f);                    // 1
  funct1(const_ptr_f);              // 2

  funct2(ptr_f);                    // 3
  funct2(const_ptr_f);              // 4

  funct3(ptr_f);                    // 5
  funct3(const_ptr_f);              // 6

  return 0;
}
```

	Valid Expression	Prints
Line 1	☐	
Line 2	☐	
Line 3	☐	
Line 4	☐	
Line 5	☐	
Line 6	☐	

```cpp
struct Foo{};

void funct1(Foo *) { printf("funct1\n"); }
void funct2(Foo const *) { printf("funct2\n"); }
void funct3(Foo *) { printf("funct3(*)\n"); }
void funct3(Foo const *) { printf("funct3(const*)\n"); }

int main()
{
  Foo f{};
  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  funct1(ptr_f);                    // 1
  funct1(const_ptr_f);              // 2

  funct2(ptr_f);                    // 3
  funct2(const_ptr_f);              // 4

  funct3(ptr_f);                    // 5
  funct3(const_ptr_f);              // 6

  return 0;
}
```

	Valid Expression	Prints
Line 1	☑	
Line 2	☐	
Line 3	☐	
Line 4	☐	
Line 5	☐	
Line 6	☐	

```
struct Foo{};

void funct1(Foo *) { printf("funct1\n"); }
void funct2(Foo const *) { printf("funct2\n"); }
void funct3(Foo *) { printf("funct3(*)\n"); }
void funct3(Foo const *) { printf("funct3(const*)\n"); }

int main()
{
  Foo f{};
  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  funct1(ptr_f);                    // 1
  funct1(const_ptr_f);              // 2

  funct2(ptr_f);                    // 3
  funct2(const_ptr_f);              // 4

  funct3(ptr_f);                    // 5
  funct3(const_ptr_f);              // 6

  return 0;
}
```

	Valid Expression	Prints
Line 1	☑	funct1
Line 2	☐	
Line 3	☐	
Line 4	☐	
Line 5	☐	
Line 6	☐	

```
struct Foo{};

void funct1(Foo *) { printf("funct1\n"); }
void funct2(Foo const *) { printf("funct2\n"); }
void funct3(Foo *) { printf("funct3(*)\n"); }
void funct3(Foo const *) { printf("funct3(const*)\n"); }

int main()
{
  Foo f{};
  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  funct1(ptr_f);                    // 1
  funct1(const_ptr_f);              // 2

  funct2(ptr_f);                    // 3
  funct2(const_ptr_f);              // 4

  funct3(ptr_f);                    // 5
  funct3(const_ptr_f);              // 6

  return 0;
}
```

	Valid Expression	Prints
Line 1	☑	`funct1`
Line 2	☒	
Line 3	☐	
Line 4	☐	
Line 5	☐	
Line 6	☐	

```
struct Foo{};

void funct1(Foo *) { printf("funct1\n"); }
void funct2(Foo const *) { printf("funct2\n"); }
void funct3(Foo *) { printf("funct3(*)\n"); }
void funct3(Foo const *) { printf("funct3(const*)\n"); }

int main()
{
  Foo f{};
  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  funct1(ptr_f);                    // 1
  funct1(const_ptr_f);              // 2

  funct2(ptr_f);                    // 3
  funct2(const_ptr_f);              // 4

  funct3(ptr_f);                    // 5
  funct3(const_ptr_f);              // 6

  return 0;
}
```

	Valid Expression	Prints
Line 1	☑	funct1
Line 2	☒	
Line 3	☐	
Line 4	☐	
Line 5	☐	
Line 6	☐	

```
struct Foo{};

void funct1(Foo *) { printf("funct1\n"); }
void funct2(Foo const *) { printf("funct2\n"); }
void funct3(Foo *) { printf("funct3(*)\n"); }
void funct3(Foo const *) { printf("funct3(const*)\n"); }

int main()
{
  Foo f{};
  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  funct1(ptr_f);                    // 1
  funct1(const_ptr_f);              // 2

  funct2(ptr_f);                    // 3
  funct2(const_ptr_f);              // 4

  funct3(ptr_f);                    // 5
  funct3(const_ptr_f);              // 6

  return 0;
}
```

	Valid Expression	Prints
Line 1	☑	funct1
Line 2	☒	
Line 3	☑	
Line 4	☐	
Line 5	☐	
Line 6	☐	

```
struct Foo{};

void funct1(Foo *) { printf("funct1\n"); }
void funct2(Foo const *) { printf("funct2\n"); }
void funct3(Foo *) { printf("funct3(*)\n"); }
void funct3(Foo const *) { printf("funct3(const*)\n"); }

int main()
{
  Foo f{};
  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  funct1(ptr_f);                    // 1
  funct1(const_ptr_f);              // 2

  funct2(ptr_f);                    // 3
  funct2(const_ptr_f);              // 4

  funct3(ptr_f);                    // 5
  funct3(const_ptr_f);              // 6

  return 0;
}
```

	Valid Expression	Prints
Line 1	☑	funct1
Line 2	☒	
Line 3	☑	funct2
Line 4	☐	
Line 5	☐	
Line 6	☐	

```
struct Foo{};

void funct1(Foo *) { printf("funct1\n"); }
void funct2(Foo const *) { printf("funct2\n"); }
void funct3(Foo *) { printf("funct3(*)\n"); }
void funct3(Foo const *) { printf("funct3(const*)\n"); }

int main()
{
  Foo f{};
  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  funct1(ptr_f);                    // 1
  funct1(const_ptr_f);              // 2

  funct2(ptr_f);                    // 3
  funct2(const_ptr_f);              // 4

  funct3(ptr_f);                    // 5
  funct3(const_ptr_f);              // 6

  return 0;
}
```

	Valid Expression	Prints
Line 1	☑	funct1
Line 2	☒	
Line 3	☑	funct2
Line 4	☑	
Line 5	☐	
Line 6	☐	

```
struct Foo{};

void funct1(Foo *) { printf("funct1\n"); }
void funct2(Foo const *) { printf("funct2\n"); }
void funct3(Foo *) { printf("funct3(*)\n"); }
void funct3(Foo const *) { printf("funct3(const*)\n"); }

int main()
{
  Foo f{};
  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  funct1(ptr_f);                    // 1
  funct1(const_ptr_f);              // 2

  funct2(ptr_f);                    // 3
  funct2(const_ptr_f);              // 4

  funct3(ptr_f);                    // 5
  funct3(const_ptr_f);              // 6

  return 0;
}
```

	Valid Expression	Prints
Line 1	☑	funct1
Line 2	☒	
Line 3	☑	funct2
Line 4	☑	funct2
Line 5	☐	
Line 6	☐	

```
struct Foo{};

void funct1(Foo *) { printf("funct1\n"); }
void funct2(Foo const *) { printf("funct2\n"); }
void funct3(Foo *) { printf("funct3(*)\n"); }
void funct3(Foo const *) { printf("funct3(const*)\n"); }

int main()
{
  Foo f{};
  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  funct1(ptr_f);                    // 1
  funct1(const_ptr_f);              // 2

  funct2(ptr_f);                    // 3
  funct2(const_ptr_f);              // 4

  funct3(ptr_f);                    // 5
  funct3(const_ptr_f);              // 6

  return 0;
}
```

	Valid Expression	Prints
Line 1	☑	funct1
Line 2	☒	
Line 3	☑	funct2
Line 4	☑	funct2
Line 5	☑	
Line 6	☐	

```
struct Foo{};

void funct1(Foo *) { printf("funct1\n"); }
void funct2(Foo const *) { printf("funct2\n"); }
void funct3(Foo *) { printf("funct3(*)\n"); }
void funct3(Foo const *) { printf("funct3(const*)\n"); }

int main()
{
  Foo f{};
  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  funct1(ptr_f);                    // 1
  funct1(const_ptr_f);              // 2

  funct2(ptr_f);                    // 3
  funct2(const_ptr_f);              // 4

  funct3(ptr_f);                    // 5
  funct3(const_ptr_f);              // 6

  return 0;
}
```

	Valid Expression	Prints
Line 1	☑	funct1
Line 2	☒	
Line 3	☑	funct2
Line 4	☑	funct2
Line 5	☑	funct3(*)
Line 6	☐	

```
struct Foo{};

void funct1(Foo *) { printf("funct1\n"); }
void funct2(Foo const *) { printf("funct2\n"); }
void funct3(Foo *) { printf("funct3(*)\n"); }
void funct3(Foo const *) { printf("funct3(const*)\n"); }

int main()
{
  Foo f{};
  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  funct1(ptr_f);                    // 1
  funct1(const_ptr_f);              // 2

  funct2(ptr_f);                    // 3
  funct2(const_ptr_f);              // 4

  funct3(ptr_f);                    // 5
  funct3(const_ptr_f);              // 6

  return 0;
}
```

	Valid Expression	Prints
Line 1	☑	funct1
Line 2	☒	
Line 3	☑	funct2
Line 4	☑	funct2
Line 5	☑	funct3(*)
Line 6	☑	

```
struct Foo{};

void funct1(Foo *) { printf("funct1\n"); }
void funct2(Foo const *) { printf("funct2\n"); }
void funct3(Foo *) { printf("funct3(*)\n"); }
void funct3(Foo const *) { printf("funct3(const*)\n"); }

int main()
{
  Foo f{};
  Foo * ptr_f = &f;
  Foo const * const_ptr_f = &f;

  funct1(ptr_f);                    // 1
  funct1(const_ptr_f);              // 2

  funct2(ptr_f);                    // 3
  funct2(const_ptr_f);              // 4

  funct3(ptr_f);                    // 5
  funct3(const_ptr_f);              // 6

  return 0;
}
```

	Valid Expression	Prints
Line 1	☑	funct1
Line 2	☒	
Line 3	☑	funct2
Line 4	☑	funct2
Line 5	☑	funct3(*)
Line 6	☑	funct3(const*)

The Importance of Being `const`

- member functions may be marked `const`

```
class T
{
  returnValue FunctionName(args) CV-qualifiers;
};
```

CV-qualifiers = { const , volatile }

The Importance of Being `const`

```
class Foo
{
  int mValue = 0;
public:
  int GetValue() const;
};
```

```
int Foo::GetValue() const
{
  return mValue;
}
```

```
Foo f;
auto v = f.GetValue();
```

The Importance of Being `const`

```
int Foo::GetValue() const
{
  return mValue;
}
```

```
Foo f;
auto v = f.GetValue();
```

A pointer the **`this`** object is added as the first argument

```
int Foo::GetValue() const
{
  return mValue;
}
```

```
Foo f;
auto v = f.GetValue();
```

A pointer the `this` object is added as the first argument

```
int Foo::GetValue(Foo* const this) const
{
  return mValue;
}
```

```
Foo f;
auto v = f.GetValue();
```

The CV-qualify references the `this` object

```
int Foo::GetValue(Foo* const this) const
{
  return mValue;
}
```

```
Foo f;
auto v = f.GetValue();
```

The CV-qualify references the `this` object

```
int Foo::GetValue(Foo const* const this)
{
  return mValue;
}
```

```
Foo f;
auto v = f.GetValue();
```

The CV-qualify references the `this` object

```
int Foo::GetValue(Foo const* const this)
{
  return mValue;
}
```

For function invocation, the object is moved to first argument

```
Foo f;
auto v = f.GetValue();
```

The CV-qualify references the `this` object

```
int Foo::GetValue(Foo const* const this)
{
  return mValue;
}
```

For function invocation, the object is moved to first argument

```
Foo f;
auto v = GetValue(&f);
```

All member variables are prefixed to use the `this` object

```
int Foo::GetValue(Foo const* const this)
{
  return mValue;
}
```

```
Foo f;
auto v = GetValue(&f);
```

All member variables are prefixed to use the `this` object

```
int Foo::GetValue(Foo const* const this)
{
  return this->mValue;
}
```

```
Foo f;
auto v = GetValue(&f);
```

The compiler translates the function to a free function with a implementation defined “name mangled” version

```
int Foo::GetValue(Foo const* const this)
{
  return this->mValue;
}
```

```
Foo f;
auto v = GetValue(&f);
```

The Importance of Being `const`

The compiler translates the function to a free function with a implementation defined “name mangled” version

```
int __ZNK3Foo8GetValueEv(Foo const* const this)
{
  return this->mValue;
}
```

For function invocation, the function call is also translated

```
Foo f;
auto v = GetValue(&f);
```

The Importance of Being `const`

The compiler translates the function to a free function with a implementation defined “name mangled” version

```
int __ZNK3Foo8GetValueEv(Foo const* const this)
{
  return this->mValue;
}
```

For function invocation, the function call is also translated

```
Foo f;
auto v = __ZNK3Foo8GetValueEv(&f);
```

The Importance of Being `const`

Before

```
int Foo::GetValue() const
{
  return mValue;
}
```

After

```
int __ZNK3Foo8GetValueEv(Foo const* const this)
{
  return this->mValue;
}
```

The Importance of Being `const`

- Marking a member function as `const` means that the this pointer is to `T const`.
- The compiler will only allow expressions will not modify values of the object.

```
int __ZNK3Foo8GetValueEv(Foo const* const this)
{
  return this->mValue;
}
```

The Importance of Being `const`

- Invariants : conditions that must be true before and after interacting with an object
- Often are reflected in the state of it’s member variables.
- `const` restricts the expressions we can write an help us maintain invariants.

The Importance of Being `const`

```
class BadResourceHandler
{
public:
  BadResourceHandler(Resource * r)
     : myResource(r) {}
  ~BadResourceHandler(); // cleanup
  void DoSomething()
  {
    /// some stuff


  }


private:
  Resource * myResource;
};
```

The Importance of Being `const`

```
class BadResourceHandler
{
public:
  BadResourceHandler(Resource * r)
     : myResource(r) {}
  ~BadResourceHandler(); // cleanup
  void DoSomething()
  {
    /// some stuff
    ++myResource; // OOPS!

  }


private:
  Resource * myResource;
};
```

- Imagine over time somebody accidentally does something bad
- Invariants violated! `myResources` should be valid before and after the function.

The Importance of Being `const`

```
class BadResourceHandler
{
public:
  BadResourceHandler(Resource * r)
     : myResource(r) {}
  ~BadResourceHandler(); // cleanup
  void DoSomething() const
  {
    /// some stuff
    ++myResource; // ERROR!

  }


private:
  Resource * myResource;
};
```

- Errors caught at compile time!

The Importance of Being `const`

```
class BadResourceHandler
{
public:
  BadResourceHandler(Resource * r)
     : myResource(r) {}
  ~BadResourceHandler(); // cleanup
  void DoSomething() const
  {
    /// some stuff
    // ++myResource;
    DoMoreWork(); // ERROR!
  }
  void DoMoreWork();

private:
  Resource * myResource;
};
```

- But `const` is infectious
- “Do I really need to add const to everything?”

The Importance of Being `const`

```
class BadResourceHandler
{
public:
  BadResourceHandler(Resource * r)
     : myResource(r) {}
  ~BadResourceHandler(); // cleanup
  void DoSomething() const
  {
    /// some stuff
    // ++myResource;
    DoMoreWork();
  }
  void DoMoreWork() const;

private:
  Resource * myResource;
};
```

- But `const` is infectious
- “Do I really need to add const to everything?”
- Absolutely!

quiz_const_overload_1.cpp

```
$ make quiz_const_overload_1
$ ./quiz_const_overload_1
```

```cpp
#include <iostream>
using std::cout;

class Foo
{
public:
  void func() {
    cout << "calling non-const\n";
  }
  void func() const {
    cout << "calling const\n";
  }
};

int main()
{
  Foo a;
  Foo const b(a);
  a.func();
  b.func();
  return 0;
}
```

quiz_const_overload_1.cpp

```
$ make quiz_const_overload_1
$ ./quiz_const_overload_1
```

```cpp
#include <iostream>
using std::cout;

class Foo
{
public:
  void func() {
    cout << "calling non-const\n";
  }
  void func() const {
    cout << "calling const\n";
  }
};

int main()
{
  Foo a;
  Foo const b(a);
  a.func();
  b.func();
  return 0;
}
```

```
calling non-const
calling const
```

quiz_const_overload_2.cpp

```
$ make quiz_const_overload_2
$ ./quiz_const_overload_2
```

```cpp
#include <iostream>
using std::cout;

class Foo
{
public:
  void func() const {
    cout << "calling const\n";
  }
};

int main()
{
  Foo a;
  Foo const b(a);
  a.func();
  b.func();
  return 0;
}
```

quiz_const_overload_2.cpp

```
$ make quiz_const_overload_2
$ ./quiz_const_overload_2
```

```cpp
#include <iostream>
using std::cout;

class Foo
{
public:
  void func() const {
    cout << "calling const\n";
  }
};

int main()
{
  Foo a;
  Foo const b(a);
  a.func();
  b.func();
  return 0;
}
```

```
calling const
calling const
```

quiz_const_overload_2.cpp

```
$ make quiz_const_overload_2
$ ./quiz_const_overload_2
```

```cpp
#include <iostream>
using std::cout;

class Foo
{
public:
  void func() const {
    cout << "calling const\n";
  }
};

int main()
{
  Foo a;
  Foo const b(a);
  a.func();
  b.func();
  return 0;
}
```

```
calling const
calling const
```

```cpp
void __ZNK3Foo4funcEv(Foo const * this); // compiler generated

Foo a;
__ZNK3Foo4funcEv(&a); // arg is Foo *, converts to Foo const *
```

quiz_const_overload_3.cpp

```
$ make quiz_const_overload_3
$ ./quiz_const_overload_3
```

```cpp
#include <iostream>
using std::cout;

class Foo
{
public:
  void func() {
    cout << "calling non-const\n";
  }
};

int main()
{
  Foo a;
  Foo const b(a);
  a.func();
  b.func();
  return 0;
}
```

quiz_const_overload_3.cpp

```
$ make quiz_const_overload_3
$ ./quiz_const_overload_3
```

```cpp
#include <iostream>
using std::cout;

class Foo
{
public:
  void func() {
    cout << "calling non-const\n";
  }
};

int main()
{
  Foo a;
  Foo const b(a);
  a.func();
  b.func();
  return 0;
}
```

```
ERROR: cannot compile
```

quiz_const_overload_3.cpp

```
$ make quiz_const_overload_3
$ ./quiz_const_overload_3
```

```cpp
#include <iostream>
using std::cout;

class Foo
{
public:
  void func() {
    cout << "calling non-const\n";
  }
};

int main()
{
  Foo a;
  Foo const b(a);
  a.func();
  b.func();
  return 0;
}
```

```
ERROR: cannot compile
```

```cpp
void __ZN3Foo4funcEv(Foo * this); // compiler generated

Foo const b(a);
__ZN3Foo4funcEv(&b); // arg is Foo const *, cannot convert to Foo *
```

```
int data;

struct Foo
{
  void Function() const
  {
    value++;            // 1
    a++;                // 2
    b++;                // 3
    *a = 20;            // 4
    *b = 20;            // 5
    refData = 20;       // 6
  }
  int value = 0;
  int * a = &data;
  int const * b = &data;
  int & refData = data;
};
```

Valid Expression

Line 1 ☐

Line 2 ☐

Line 3 ☐

Line 4 ☐

Line 5 ☐

Line 6 ☐

```
int data;

struct Foo
{
  void Function() const
  {
    value++;            // 1
    a++;                // 2
    b++;                // 3
    *a = 20;            // 4
    *b = 20;            // 5
    refData = 20;       // 6
  }
  int value = 0;
  int * a = &data;
  int const * b = &data;
  int & refData = data;
};
```

Valid Expression

Line 1 ☒

Line 2 ☐

Line 3 ☐

Line 4 ☐

Line 5 ☐

Line 6 ☐

```
int data;

struct Foo
{
  void Function() const
  {
    value++;            // 1
    a++;                // 2
    b++;                // 3
    *a = 20;            // 4
    *b = 20;            // 5
    refData = 20;       // 6
  }
  int value = 0;
  int * a = &data;
  int const * b = &data;
  int & refData = data;
};
```

Valid Expression

Line 1 ☒

Line 2 ☒

Line 3 ☐

Line 4 ☐

Line 5 ☐

Line 6 ☐

```
int data;

struct Foo
{
  void Function() const
  {
    value++;            // 1
    a++;                // 2
    b++;                // 3
    *a = 20;            // 4
    *b = 20;            // 5
    refData = 20;       // 6
  }
  int value = 0;
  int * a = &data;
  int const * b = &data;
  int & refData = data;
};
```

Valid Expression

Line 1 ☒

Line 2 ☒

Line 3 ☒

Line 4 ☐

Line 5 ☐

Line 6 ☐

```
int data;

struct Foo
{
  void Function() const
  {
    value++;            // 1
    a++;                // 2
    b++;                // 3
    *a = 20;            // 4
    *b = 20;            // 5
    refData = 20;       // 6
  }
  int value = 0;
  int * a = &data;
  int const * b = &data;
  int & refData = data;
};
```

Valid Expression

Line 1 ☒

Line 2 ☒

Line 3 ☒

Line 4 ☑

Line 5 ☐

Line 6 ☐

```
int data;

struct Foo
{
  void Function() const
  {
    value++;            // 1
    a++;                // 2
    b++;                // 3
    *a = 20;            // 4
    *b = 20;            // 5
    refData = 20;       // 6
  }
  int value = 0;
  int * a = &data;
  int const * b = &data;
  int & refData = data;
};
```

Valid Expression

Line 1 ☒

Line 2 ☒

Line 3 ☒

Line 4 ☑

Line 5 ☒

Line 6 ☐

```
int data;

struct Foo
{
  void Function() const
  {
    value++;            // 1
    a++;                // 2
    b++;                // 3
    *a = 20;            // 4
    *b = 20;            // 5
    refData = 20;       // 6
  }
  int value = 0;
  int * a = &data;
  int const * b = &data;
  int & refData = data;
};
```

Valid Expression

Line 1 ☒

Line 2 ☒

Line 3 ☒

Line 4 ☑

Line 5 ☒

Line 6 ☑

```
int data;

struct Foo
{
  void Function() const
  {
    value++;              // 1
    a++;                  // 2
    b++;                  // 3
    *a = 20;              // 4
    *b = 20;              // 5
    refData = 20;         // 6
  }
  int value = 0;
  int * a = &data;
  int const * b = &data;
  int & refData = data;
};
```

Valid Expression

Line 1 ☐

Line 2 ☐

Line 3 ☐

Line 4 ☑

Line 5 ☐

Line 6 ☑

Bit-wise `const` vs Logical `const`

- **Bit-wise `const`**:
  - A member function is **Bit-wise `const`** if it doesn't modify any of the bits inside the object
  - This is the compiler's view - "a `const` member function isn't allowed to modify any of the non-static data members of the object on which it is invoked."

Bit-wise `const` vs Logical `const`

- **Logical `const`**:
  - **Logical `const`** means that from the client's point of view, the function has not changed anything.
  - There are no detectable changes.
  - Observably `const`.

```
int data;

struct Foo
{
  void Function() const
  {
    value++;            // 1
    a++;                // 2
    b++;                // 3
    *a = 20;            // 4
    *b = 20;            // 5
    refData = 20;       // 6
  }
  int value = 0;
  int * a = &data;
  int const * b = &data;
  int & refData = data;
};
```

- What do we mean when we mark a member function `const`?
- `const` member functions frequently are “observer” functions
- You should strive to make your member functions **`Observably const`**.

The Importance of Being `const`

- What do you do when you need to do something non-`const` in a `const` member function?

The Importance of Being `const`

- Caching or logging are prime examples and valid use cases.
- `const` functions cannot modify object

```
class DataHolder {
public:
  int GetCheckSum() const {
    ++mTimesCalled;
    return CalculateChecksum();
  }
  void AddMore(Data const& d) {
    // Modify the Data
  }
private:
  int CalculateChecksum() const;

  int mTimesCalled{0};
};

// ERROR! Does not compile
void DoSomeWork(DataHolder const& d) {
  auto cksum = d.GetCheckSum();
  ;;;
}
```

`mutable`

- On these occasions you reach for **`mutable`**, the C++ wiggle room for **`const`**.

```
class DataHolder {
  ...
private:
  ...
  mutable int mTimesCalled{0};
};
```

- **`mutable`** tells the compiler that the value will change even if the function is **`const`**.

The Importance of Being `const`

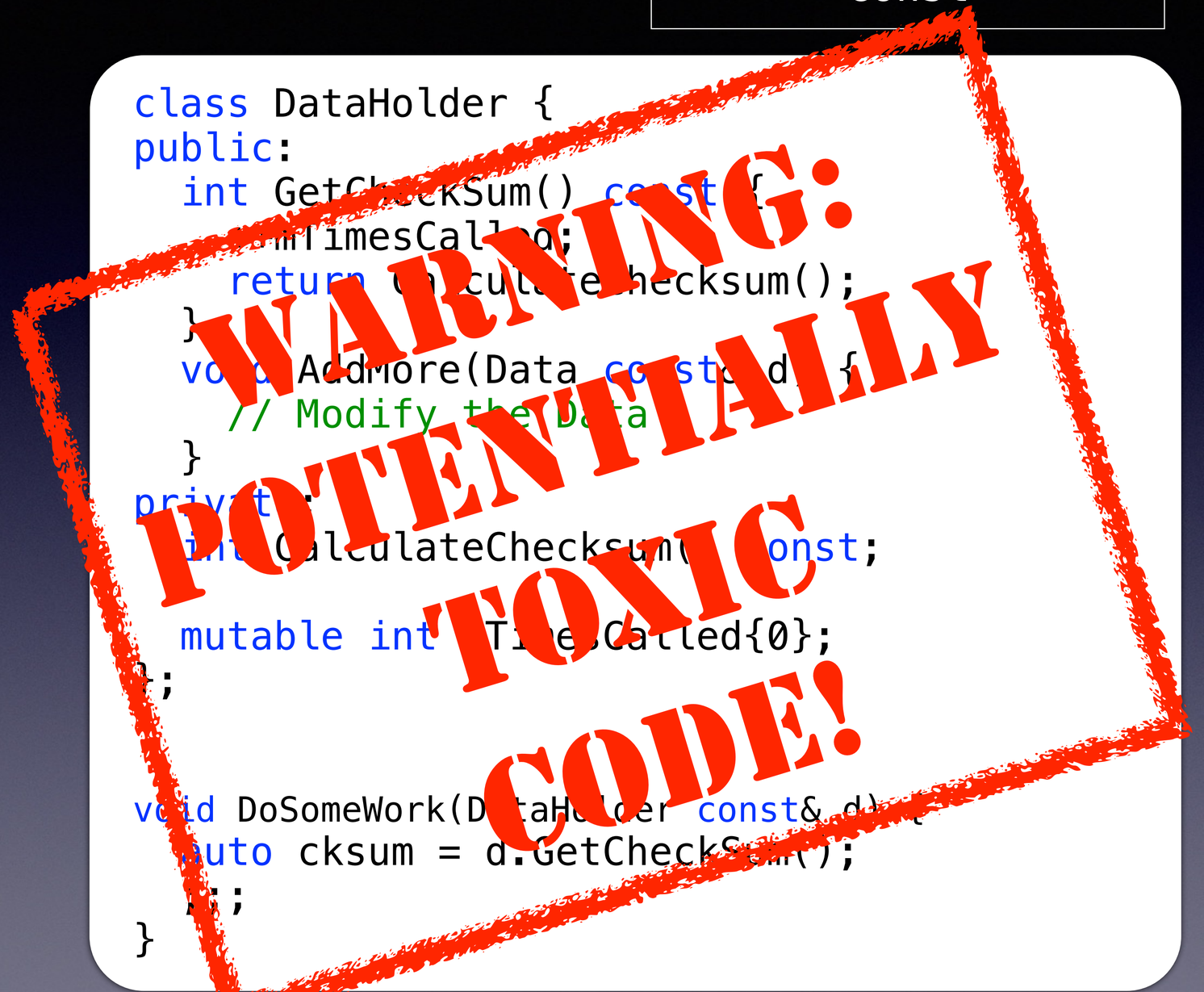
- Ok, now the program compiles.
- Are we good?

```
class DataHolder {
public:
  int GetCheckSum() const {
    ++mTimesCalled;
    return CalculateChecksum();
  }
  void AddMore(Data const& d) {
    // Modify the Data
  }
private:
  int CalculateChecksum() const;

  mutable int mTimesCalled{0};
};


void DoSomeWork(DataHolder const& d) {
  auto cksum = d.GetCheckSum();
  ;;;
}
```

The Importance of Being `const`

- Ok, now the program compiles.
- Are we good?

```
class DataHolder {
public:
  int GetCheckSum() const {
    ++mTimesCalled;
    return CalculateChecksum();
  }
  void AddMore(Data const& d) {
    // Modify the Data
  }
private:
  int CalculateChecksum() const;

  mutable int mTimesCalled{0};
};


void DoSomeWork(DataHolder const& d) {
  auto cksum = d.GetCheckSum();
  ...;
}
```



The Importance of Being `const`

```
class DataHolder {
public:
  int GetCheckSum() const {
    ++mTimesCalled;
    return CalculateChecksum();
  }
  void AddMore(Data const& d) {
    // Modify the Data
  }
private:
  int CalculateChecksum() const;

  mutable int mTimesCalled{0};
};


void DoSomeWork(DataHolder const& d) {
  auto cksum = d.GetCheckSum();
  ;;;
}
```

The Importance of Being `const`

- This code contains subtle bugs.

```
class DataHolder {
public:
  int GetCheckSum() const {
    ++mTimesCalled;
    return CalculateChecksum();
  }
  void AddMore(Data const& d) {
    // Modify the Data
  }
private:
  int CalculateChecksum() const;

  mutable int mTimesCalled{0};
};


void DoSomeWork(DataHolder const& d) {
  auto cksum = d.GetCheckSum();
  ;;;
}
```

The Importance of Being `const`

- This code contains subtle bugs.
- Why?

```
class DataHolder {
public:
  int GetCheckSum() const {
    ++mTimesCalled;
    return CalculateChecksum();
  }
  void AddMore(Data const& d) {
    // Modify the Data
  }
private:
  int CalculateChecksum() const;

  mutable int mTimesCalled{0};
};


void DoSomeWork(DataHolder const& d) {
  auto cksum = d.GetCheckSum();
  ;;;
}
```

The Importance of Being `const`

- This code contains subtle bugs.
- Why?
- Threads

```
class DataHolder {
public:
  int GetCheckSum() const {
    ++mTimesCalled;
    return CalculateChecksum();
  }
  void AddMore(Data const& d) {
    // Modify the Data
  }
private:
  int CalculateChecksum() const;

  mutable int mTimesCalled{0};
};


void DoSomeWork(DataHolder const& d) {
  auto cksum = d.GetCheckSum();
  ;;;
}
```

The Importance of Being `const`

- What is a data-race?
- “The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.” ISO N3690: § 1.10.20 [intro.multithread]
- If two or more threads access the same memory location without synchronization, and at least one is a writer, you have undefined behavior.

`const` means data-race free

- If two or more threads access the same memory location without synchronization, and at least one is a writer, you have undefined behavior.

- Let’s walk it through

```
DataHolder gData;
```

Thread 1

```
// Reading gData
// Calling const member function
auto cksum = gData.GetCheckSum();
```

Thread 2

```
// Reading gData
// Calling const member function
auto checksum = gData.GetCheckSum();
```

- Let’s walk it through

```
DataHolder gData;
```

```
// gData.mTimesCalled == 0
```

Thread 1

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  ++mTimesCalled;
  return CalculateChecksum();
}
```

Thread 2

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  ++mTimesCalled;
  return CalculateChecksum();
}
```

- Let’s walk it through

```
DataHolder gData;
```

```
// gData.mTimesCalled == 0
```

Thread 1

```
// Reading gData
// and
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

Thread 2

```
// Reading gData
// and
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

- Let’s walk it through

```
DataHolder gData;
```

```
// gData.mTimesCalled == 0
```

Thread 1

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

Thread 2

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

- Both threads read **`mTimeCalled`** to local register

- Let’s walk it through

```
DataHolder gData;
```

```
// gData.mTimesCalled == 0
```

Thread 1

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

Thread 2

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

- Both threads read **`mTimeCalled`** to local register

- Let’s walk it through

```
DataHolder gData;
```

```
// gData.mTimesCalled == 0
```

Thread 1

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

Thread 2

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

- Both threads increment their registers.

- Let’s walk it through

```
DataHolder gData;
```

```
// gData.mTimesCalled == 0
```

Thread 1

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

Thread 2

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

- Both threads increment their registers.

- Let’s walk it through

```
DataHolder gData;
```

```
// gData.mTimesCalled == 1
```

Thread 1

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

Thread 2

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

- Both threads write `(mTimeCalled+1)` to `mTimeCalled`

- Let’s walk it through

```
DataHolder gData;
```

```
// gData.mTimesCalled == 1
```

Thread 1

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

Thread 2

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

- Both threads write `(mTimeCalled+1)` to `mTimeCalled`

- Let’s walk it through

```
DataHolder gData;
```

```
// gData.mTimesCalled == 1
```

Thread 1

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

Thread 2

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

- `mTimesCalled` only incremented by 1!!!

The Importance of Being `const`

- The fix is “easy”
- Add synchronization primitives around your data hazards.

```
class DataHolder {
public:
  int GetCheckSum() const {
    std::lock_guard<std::mutex> l(mMutex);
    ++mTimesCalled;
    return CalculateChecksum();
  }
  void AddMore(Data const& d) {
    // Modify the Data
  }
private:
  int CalculateChecksum() const;

  mutable int mTimesCalled{0};
  mutable std::mutex mMutex;
};
```

- What about this?

```
DataHolder gData;
```

```
// gData.mTimesCalled == 0
```

Thread 1

```
// Reading gData
// ...
a
```

```
int GetCheckSum() const {
  std::lock_guard<std::mutex> l(mMutex);
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

Thread 2

```
// Reading gData
// ...
a
```

```
int GetCheckSum() const {
  std::lock_guard<std::mutex> l(mMutex);
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

```
DataHolder gData;
```

```
// gData.mTimesCalled == 0
```

Thread 1

```
// Reading gData
// 
a
```

```
int GetCheckSum() const {
  std::lock_guard<std::mutex> l(mMutex);
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

Thread 2

```
// Reading gData
// 
a
```

```
int GetCheckSum() const {
  std::lock_guard<std::mutex> l(mMutex);
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

- Thread 1 goes first, grabs the lock

```
DataHolder gData;
```

```
// gData.mTimesCalled == 0
```

Thread 1

```
int GetCheckSum() const {
  std::lock_guard<std::mutex> l(mMutex);
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

Thread 2

```
int GetCheckSum() const {
  std::lock_guard<std::mutex> l(mMutex);
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

- Thread 2 proceeds, but cannot get the lock

```
DataHolder gData;
```

```
// gData.mTimesCalled == 1
```

Thread 1

```
int GetCheckSum() const {
  std::lock_guard<std::mutex> l(mMutex);
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

Thread 2

```
int GetCheckSum() const {
  std::lock_guard<std::mutex> l(mMutex);
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

- Thread 1 finishes, release the lock.
- Now Thread 2 can proceed

```
DataHolder gData;
```

```
// gData.mTimesCalled == 2
```

Thread 1

```
int GetCheckSum() const {
  std::lock_guard<std::mutex> l(mMutex);
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

Thread 2

```
int GetCheckSum() const {
  std::lock_guard<std::mutex> l(mMutex);
  mTimesCalled = mTimesCalled + 1;
  return CalculateChecksum();
}
```

- Thread 2 finishes.
- `mTimesCalled` now correctly incremented.

The Importance of Being `const`

- `mutex`s are the basic synchronization mechanism, but they tend to be heavy weight
- Perfect place for `atomic`s
- “Objects of `atomic` types are the only C++ objects that are free from data races; that is, if one thread writes to an `atomic` object while another thread reads from it, the behavior is well-defined.” - cppreference.com

```
class DataHolder {
public:
  int GetCheckSum() const {
    ++mTimesCalled;
    return CalculateChecksum();
  }
  void AddMore(Data const& d) {
    // Modify the Data
  }
private:
  int CalculateChecksum() const;

  mutable std::atomic<int> mTimesCalled{0};
};
```

- With atomics

```
DataHolder gData;
```

```
// gData.mTimesCalled == 0
```

Thread 1

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  ++mTimesCalled;
  return CalculateChecksum();
}
```

Thread 2

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  ++mTimesCalled;
  return CalculateChecksum();
}
```

- With atomics

```
DataHolder gData;
```

```
// gData.mTimesCalled == 1
```

Thread 1

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  ++mTimesCalled;
  return CalculateChecksum();
}
```

Thread 2

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  ++mTimesCalled;
  return CalculateChecksum();
}
```

- Atomic increments are well defined on different threads

- With atomics

```
DataHolder gData;
```

```
// gData.mTimesCalled == 2
```

Thread 1

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  ++mTimesCalled;
  return CalculateChecksum();
}
```

Thread 2

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  ++mTimesCalled;
  return CalculateChecksum();
}
```

- Atomic increments are well defined on different threads

- With atomics

```
DataHolder gData;
```

```
// gData.mTimesCalled == 2
```

Thread 1

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  ++mTimesCalled;
  return CalculateChecksum();
}
```

Thread 2

```
// Reading gData
// and
a
```

```
int GetCheckSum() const {
  ++mTimesCalled;
  return CalculateChecksum();
}
```

- `mTimesCalled` correctly incremented.

- `mutex`s and `atomic`s are not CopyConstructible.
- This means objects that use them are not copiable (but they are moveable)
- Be aware that this may have wide spread changes in your code.

```
class DataHolder;

int main(int argc, char* argv[])
{
  DataHolder d;
  DataHolder d1(d);
}
```

```
test4.cpp:33:13: error: call to implicitly-
deleted copy constructor of 'DataHolder'
        DataHolder d1(d);
                   ^  ~
test4.cpp:24:22: note: copy constructor of
'DataHolder' is implicitly deleted because field
'mMutex' has an inaccessible copy constructor
  mutable std::mutex mMutex;
                     ^
```

The Importance of Being `const`

```
class DataHolder {
public:
  int GetCheckSum() const {
    return CalculateChecksum();
  }
  void AddMore(Data const& d) {
    // Modify the Data
  }
private:
  int CalculateChecksum() const;
};



void DoSomeWork(DataHolder const& d) {
  auto cksum = d.GetCheckSum();
  ;;;
}
```

- What if `CalculateChecksum` was very expensive. How could we speed things up?

The Importance of Being `const`

```
class DataHolder {
public:
  int GetCheckSum() const {
    if (!mCached) {
      mCached = true;
      mCachedChecksum = CalculateChecksum();
    }
    return mCachedChecksum;
  }
  void AddMore(Data const& d) {
    mCached = false;
    // Modify the Data
  }
private:
  int CalculateChecksum() const;

  int mCachedChecksum{0};
  bool mCached{false};
};

// ERROR! Does not compile
void DoSomeWork(DataHolder const& d) {
  auto cksum = d.GetCheckSum();
  ;;;
}
```

- Caching or logging are prime examples and valid use cases.
- But that lousy `const` is getting in the way!

The Importance of Being `const`

```
class DataHolder {
public:
  int GetCheckSum() const {
    if (!mCached) {
      mCached = true;
      mCachedChecksum = CalculateChecksum();
    }
    return mCachedChecksum;
  }
  void AddMore(Data const& d) {
    mCached = false;
    // Modify the Data
  }
private:
  int CalculateChecksum() const;

  mutable int mCachedChecksum{0};
  mutable bool mCached{false};
};

void DoSomeWork(DataHolder const& d) {
  auto cksum = d.GetCheckSum();
  ;;;
}
```

- Ok, now the program compiles.
- Are we good?

The Importance of Being `const`

```
class DataHolder {
public:
  int GetCheckSum() const {
    if (!mCached) {
      mCached = true;
      mCachedChecksum = CalculateChecksum();
    }
    return mCachedChecksum;
  }
  void AddMore(Data const& d) {
    mCached = false;
    // Modify the Data
  }
private:
  int CalculateChecksum() const;

  mutable int mCachedChecksum{0};
  mutable bool mCached{false};
};

void DoSomeWork(DataHolder const& d) {
  auto cksum = d.GetCheckSum();
  // ...;
}
```

WARNING: POTENTIALLY TOXIC CODE!

- Ok, now the program compiles.
- Are we good?

The Importance of Being `const`

```
class DataHolder {
public:
  DataHolder() = default;
  DataHolder(const DataHolder& d) :
    mCachedChecksum{d.mCachedChecksum.load()},
    mCached{d.mCached.load()}
  { /* other work to copy Data */ }

  int GetCheckSum() const {
    if (!mCached) {
      mCached = true;
      mCachedChecksum = CalculateChecksum();
    }
    return mCachedChecksum;
  }
  void AddMore(Data const& d) {
    mCached = false;
    // Modify the Data
  }
private:
  int CalculateChecksum() const;
  mutable std::atomic<int> mCachedChecksum{0};
  mutable std::atomic<bool> mCached{false};
};
```

- Right, I'll use `atomics` here.
- And I'll add copy constructor because atomics have are not CopyConstructable.

The Importance of Being `const`

```
class DataHolder {
public:
  DataHolder() = default;
  DataHolder(const DataHolder& d) :
    mCachedChecksum{d.mCachedChecksum.load()},
    mCached{d.mCached.load()}
  { /* other work to copy Data */ }

  int GetCheckSum() const {
    if (!mCached) {
      mCached = true;
      mCachedChecksum = CalculateChecksum();
    }
    return mCachedChecksum;
  }
  void AddMore(Data const& d) {
    mCached = false;
    // Modify the Data
  }
private:
  int CalculateChecksum() const;
  mutable std::atomic<int> mCachedChecksum{0};
  mutable std::atomic<bool> mCached{false};
};
```

- Am I good to go?

The Importance of Being `const`

```
class DataHolder {
public:
  DataHolder() = default;
  DataHolder(const DataHolder& d) :
    mCachedChecksum{d.mCachedChecksum.load()},
    mCached{d.mCached.load()}
  { /* other work to copy Data */ }

  int GetCheckSum() const {
    if (!mCached) {
      mCached = true;
      mCachedChecksum = CalculateChecksum();
    }
    return mCachedChecksum;
  }
  void AddMore(Data const& d) {
    mCached = false;
    // Modify the Data
  }
private:
  int CalculateChecksum() const;
  mutable std::atomic<int> mCachedChecksum{0};
  mutable std::atomic<bool> mCached{false};
};
```

- Am I good to go?

The Importance of Being `const`

```
class DataHolder {
public:
  DataHolder() = default;
  DataHolder(const DataHolder& d) :
    mCachedChecksum{d.mCachedChecksum.load()},
    mCached{d.mCached.load()}
  { /* other work to copy Data */ }

  int GetCheckSum() const {
    if (!mCached) {
      mCached = true;
      mCachedChecksum = CalculateChecksum();
    }
    return mCachedChecksum;
  }
  void AddMore(Data const& d) {
    mCached = false;
    // Modify the Data
  }
private:
  int CalculateChecksum() const;
  mutable std::atomic<int> mCachedChecksum{0};
  mutable std::atomic<bool> mCached{false};
};
```

- Am I good to go?
- Still has a race-condition.

- What about this?

```
DataHolder gData;
```

Thread 1

```
// Reading gData
// Calling const member function
auto cksum = gData.GetCheckSum();
```

Thread 2

```
// Modifying gData
// Calling non-const member function
gData.AddMore(GetData("MyFile.txt"));
```

- Even in the original implementation, this would be a data race.
- But this is expected. Clients expect that mixing `const` and non-`const` functions is bad (unless you have specifically designed to be thread-safe)

Case 1:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 0
// gData.mCached == false
```

Thread 1

```
// Reading gData
// Calling const member function
auto cksum = gData.GetCheckSum();
```

Thread 2

```
// Reading gData
// Calling const member function
auto checksum = gData.GetCheckSum();
```

Case 1:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 0
// gData.mCached == false
```

Thread 1

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCached = true;
    mCachedChecksum = CalculateChecksum();
  }
  return mCachedChecksum;
}
```

Thread 2

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCached = true;
    mCachedChecksum = CalculateChecksum();
  }
  return mCachedChecksum;
}
```

- Both threads enter the function at the same time.
- Thread 1 proceeds

Case 1:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 0
// gData.mCached == false
```

Thread 1

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCached = true;
    mCachedChecksum = CalculateChecksum();
  }
  return mCachedChecksum;
}
```

Thread 2

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCached = true;
    mCachedChecksum = CalculateChecksum();
  }
  return mCachedChecksum;
}
```

- Assume this is the first time so `mCache` is `false`.
- Thread 1 proceeds

Case 1:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 0
// gData.mCached == true
```

Thread 1

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCached = true;
    mCachedChecksum = CalculateChecksum();
  }
  return mCachedChecksum;
}
```

Thread 2

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCached = true;
    mCachedChecksum = CalculateChecksum();
  }
  return mCachedChecksum;
}
```

- Thread 1 sets `mCached` to true.
- Now Thread 2 proceeds

Case 1:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 0
// gData.mCached == true
```

Thread 1

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCached = true;
    mCachedChecksum = CalculateChecksum();
  }
  return mCachedChecksum;
}
```

Thread 2

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCached = true;
    mCachedChecksum = CalculateChecksum();
  }
  return mCachedChecksum;
}
```

- Thread 2 sees that `mCached` is true.
- Thread 2 proceeds

Case 1:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 0
// gData.mCached == true
```

Thread 1

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCached = true;
    mCachedChecksum = CalculateChecksum();
  }
  return mCachedChecksum;
}
```

Thread 2

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCached = true;
    mCachedChecksum = CalculateChecksum();
  }
  return mCachedChecksum;
}
```

- Thread 2 sees that `mCached` is true.
- Thread 2 proceeds

Case 1:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 0
// gData.mCached == true
```

Thread 1

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCached = true;
    mCachedChecksum = CalculateChecksum();
  }
  return mCachedChecksum;
}
```

Thread 2

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCached = true;
    mCachedChecksum = CalculateChecksum();
  }
  return mCachedChecksum;
}
```

- Thread 2 returns **`mCachedChecksum`** before Thread 1 has calculated the answer.
- **`GetCheckSum`** returns incorrect value!!!

- Issue seems to be that `mCached` setting and `mCachedChecksum` are in the wrong order.

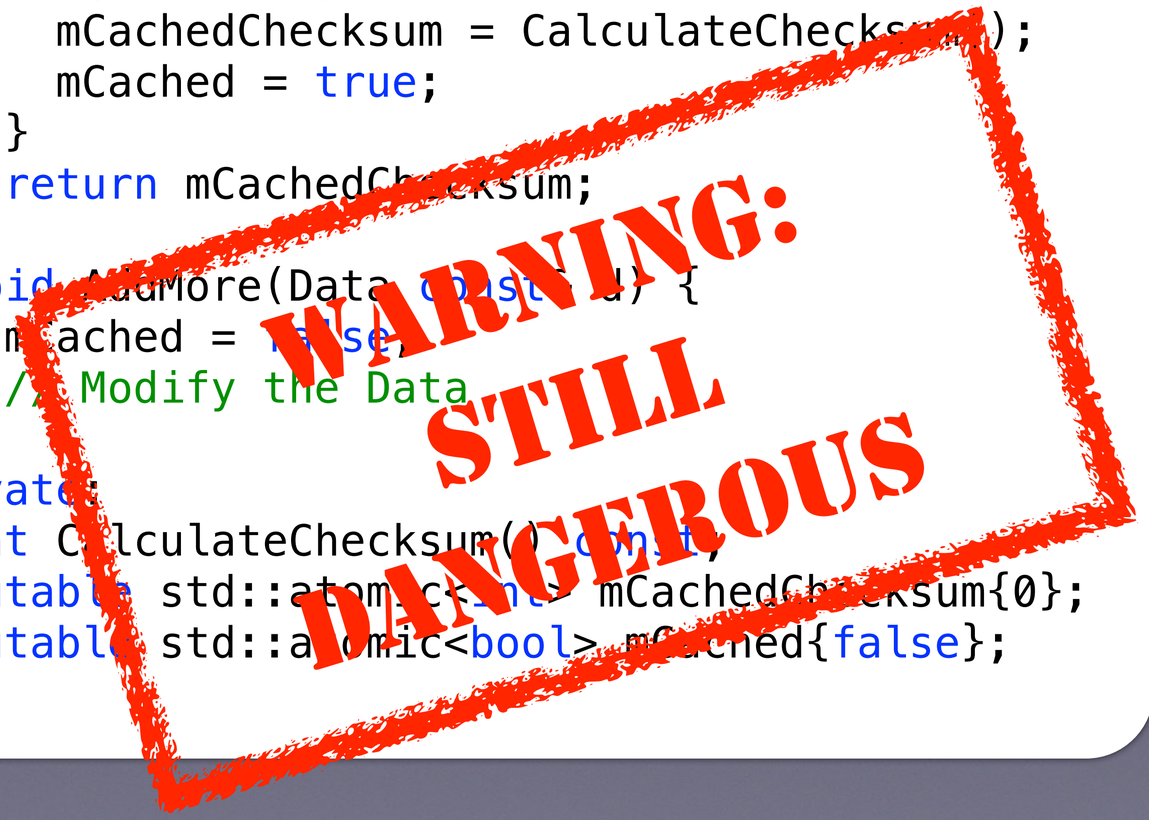
```
class DataHolder {
public:
  DataHolder() = default;
  DataHolder(const DataHolder& d) :
    mCachedChecksum{d.mCachedChecksum.load()},
    mCached{d.mCached.load()}
  { /* other work to copy Data */ }

  int GetCheckSum() const {
    if (!mCached) {
      mCached = true;
      mCachedChecksum = CalculateChecksum();
    }
    return mCachedChecksum;
  }
  void AddMore(Data const& d) {
    mCached = false;
    // Modify the Data
  }
private:
  int CalculateChecksum() const;
  mutable std::atomic<int> mCachedChecksum{0};
  mutable std::atomic<bool> mCached{false};
};
```

- Issue seems to be that `mCached` setting and `mCachedChecksum` are in the wrong order.
- Ok, how about now?

```
class DataHolder {
public:
  DataHolder() = default;
  DataHolder(const DataHolder& d) :
    mCachedChecksum{d.mCachedChecksum.load()},
    mCached{d.mCached.load()}
  { /* other work to copy Data */ }

  int GetCheckSum() const {
    if (!mCached) {
      mCachedChecksum = CalculateChecksum();
      mCached = true;
    }
    return mCachedChecksum;
  }
  void AddMore(Data const& d) {
    mCached = false;
    // Modify the Data
  }
private:
  int CalculateChecksum() const;
  mutable std::atomic<int> mCachedChecksum{0};
  mutable std::atomic<bool> mCached{false};
};
```

- Issue seems to be that **`mCached`** setting and **`mCachedChecksum`** are in the wrong order.
- Ok, how about now?

```
class DataHolder {
public:
  DataHolder() = default;
  DataHolder(const DataHolder& d) :
    mCachedChecksum{d.mCachedChecksum.load()},
    mCached{d.mCached.load()}
  { /* other work to copy Data */ }

  int GetCheckSum() const {
    if (!mCached) {
      mCachedChecksum = CalculateChecksum();
      mCached = true;
    }
    return mCachedChecksum;
  }
  void AddMore(Data const& d) {
    mCached = false;
    // Modify the Data
  }
private:
  int CalculateChecksum() const;
  mutable std::atomic<int> mCachedChecksum{0};
  mutable std::atomic<bool> mCached{false};
};
```



Case 1:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 0
// gData.mCached == false
```

Thread 1

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

Thread 2

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

- Both thread read `mCached` as `false`.

Case 1:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 42
// gData.mCached == false
```

Thread 1

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

Thread 2

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

- Both thread proceed to execute the `if` statement
- Both threads execute `CalculateChecksum()`

Case 1:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 42
// gData.mCached == false
```

Thread 1

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

Thread 2

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

- Both thread proceed to execute the `if` statement
- Both threads execute `CalculateChecksum()`

Case 1:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 42
// gData.mCached == true
```

Thread 1

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

Thread 2

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

- Writes to `atomic`s are synchronized, so no issues on both threads writing the values.
- Both threads eventual return the answer (assuming `CalculateChecksum` is also thread-safe)

Case 1:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 42
// gData.mCached == true
```

Thread 1

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

Thread 2

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

- Writes to **`atomic`**s are synchronized, so no issues on both threads writing the values.
- Both threads eventual return the answer (assuming **`CalculateChecksum`** is also thread-safe)

Case 1:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 42
// gData.mCached == true
```

Thread 1

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

Thread 2

```
// Reading gData
// Calling const member function
a
```

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

- So what's the big deal?
- More work done than needed to be.
- What if instead of 2 threads, it was 100? What if calculation was very expensive?

- There is another pernicious bug lurking in here

```
class DataHolder {
public:
  DataHolder() = default;
  DataHolder(const DataHolder& d) :
    mCachedChecksum{d.mCachedChecksum.load()},
    mCached{d.mCached.load()}
  { /* other work to copy Data */ }

  int GetCheckSum() const {
    if (!mCached) {
      mCachedChecksum = CalculateChecksum();
      mCached = true;
    }
    return mCachedChecksum;
  }
  void AddMore(Data const& d) {
    mCached = false;
    // Modify the Data
  }
private:
  int CalculateChecksum() const;
  mutable std::atomic<int> mCachedChecksum{0};
  mutable std::atomic<bool> mCached{false};
};
```

Case 1:

```
DataHolder gData;
```

Thread 1

```
// Reading gData
// Calling const member function
auto cksum = gData.GetCheckSum();
```

Thread 2

```
// Reading gData
// Calling const member function
auto checksum = gData.GetCheckSum();
```

Case 2:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 0
// gData.mCached == false
```

Thread 1

```
// Reading gData
// Calling const member function
auto cksum = gData.GetCheckSum();
```

Thread 2

```
// Reading gData
// Calling copy-ctor
auto dataCopy = gData;
```

Case 2:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 0
// gData.mCached == false
```

```
// dataCopy.mCachedChecksum == ???
// dataCopy.mCached == ???
```

Thread 1

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

Thread 2

```
DataHolder(DataHolder const& d) :
mCachedChecksum(d.mCachedChecksum),
mCached(d.mCached)
{ /* other work to copy Data */ }
```

- Assume that the CheckSum has not been set
- Thread 2 proceeds

Case 2:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 0
// gData.mCached == false
```

```
// dataCopy.mCachedChecksum == 0
// dataCopy.mCached == ???
```

Thread 1

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

Thread 2

```
DataHolder(DataHolder const& d) :
mCachedChecksum(d.mCachedChecksum),
mCached(d.mCached)
{ /* other work to copy Data */ }
```

- Thread 2 reads `mCachedChecksum`. Value is default (0).
- Thread 1 proceeds

Case 2:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 42
// gData.mCached == true
```

```
// dataCopy.mCachedChecksum == 0
// dataCopy.mCached == ???
```

Thread 1

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

Thread 2

```
DataHolder(DataHolder const& d) :
mCachedChecksum(d.mCachedChecksum),
mCached(d.mCached)
{ /* other work to copy Data */ }
```

- Thread 1 calculates `mCachedChecksum` and sets `mCached` to `true`.
- Thread 2 proceeds

Case 2:

```
DataHolder gData;
```

```
// gData.mCachedChecksum == 42
// gData.mCached == true
```

```
// dataCopy.mCachedChecksum == 0
// dataCopy.mCached == true
```

Thread 1

```
int GetCheckSum() const {
  if (!mCached) {
    mCachedChecksum = CalculateChecksum();
    mCached = true;
  }
  return mCachedChecksum;
}
```

Thread 2

```
DataHolder(DataHolder const& d) :
mCachedChecksum(d.mCachedChecksum),
mCached(d.mCached)
{ /* other work to copy Data */ }
```

- Thread 2 sets `mCached` to `true`.
- But `mCachedChecksum` value is incorrect!!!
- Race-Condition!!!

- The order of initialization in the copy constructor causes subtle issues
- This is **not** a data race.
- This happens in any case where you have two data locations that must be synchronized.

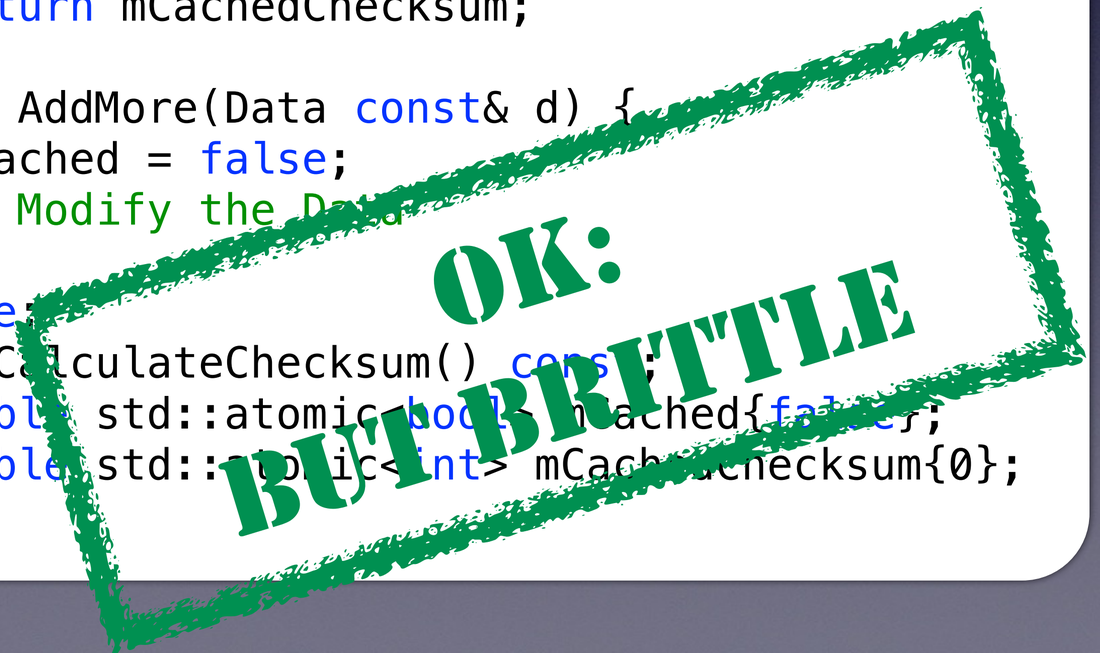
```
class DataHolder {
public:
  DataHolder() = default;
  DataHolder(const DataHolder& d) :
    mCachedChecksum{d.mCachedChecksum.load()},
    mCached{d.mCached.load()}
  { /* other work to copy Data */ }

  int GetCheckSum() const {
    if (!mCached) {
      mCachedChecksum = CalculateChecksum();
      mCached = true;
    }
    return mCachedChecksum;
  }
  void AddMore(Data const& d) {
    mCached = false;
    // Modify the Data
  }
private:
  int CalculateChecksum() const;
  mutable std::atomic<int> mCachedChecksum{0};
  mutable std::atomic<bool> mCached{false};
};
```

- Ok, how about now?

```
class DataHolder {
public:
  DataHolder() = default;
  DataHolder(const DataHolder& d) :
    mCached{d.mCached.load()}
    mCachedChecksum{d.mCachedChecksum.load()},
  { /* other work to copy Data */ }

  int GetCheckSum() const {
    if (!mCached) {
      mCachedChecksum = CalculateChecksum();
      mCached = true;
    }
    return mCachedChecksum;
  }
  void AddMore(Data const& d) {
    mCached = false;
    // Modify the Data
  }
private:
  int CalculateChecksum() const;
  mutable std::atomic<bool> mCached{false};
  mutable std::atomic<int> mCachedChecksum{0};
};
```

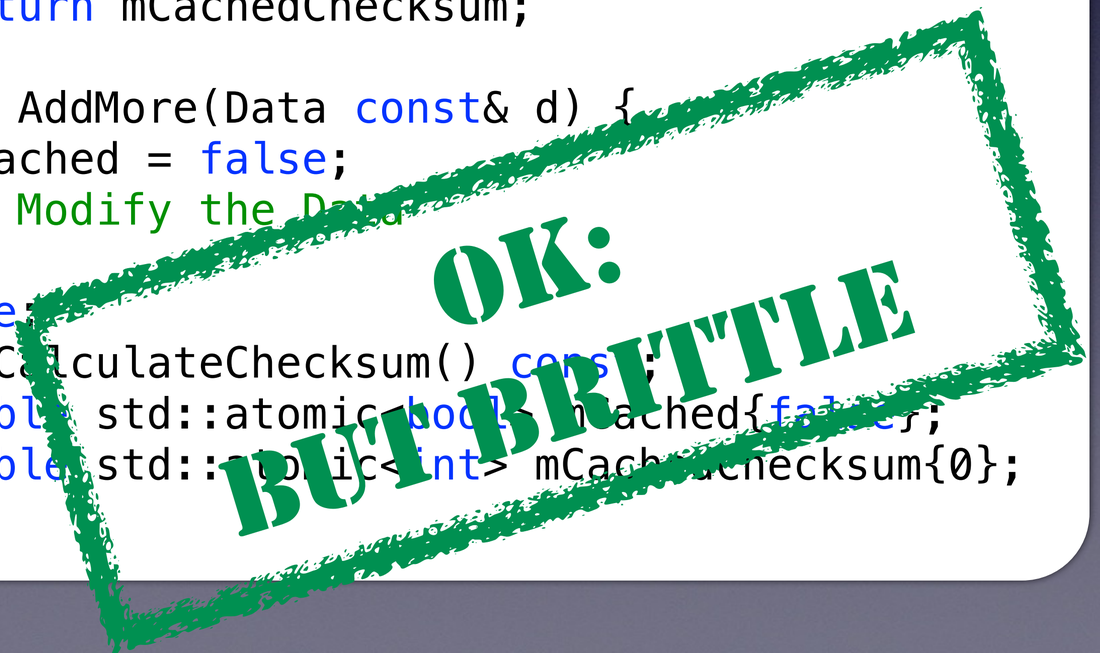
- Ok, how about now?

```
class DataHolder {
public:
  DataHolder() = default;
  DataHolder(const DataHolder& d) :
    mCached{d.mCached.load()}
    mCachedChecksum{d.mCachedChecksum.load()},
  { /* other work to copy Data */ }

  int GetCheckSum() const {
    if (!mCached) {
      mCachedChecksum = CalculateChecksum();
      mCached = true;
    }
    return mCachedChecksum;
  }
  void AddMore(Data const& d) {
    mCached = false;
    // Modify the Data
  }
private:
  int CalculateChecksum() const;
  mutable std::atomic<bool> mCached{false};
  mutable std::atomic<int> mCachedChecksum{0};
};
```



- How well will this be maintained?
- How do you test it?

```
class DataHolder {
public:
  DataHolder() = default;
  DataHolder(const DataHolder& d) :
    mCached{d.mCached.load()}
    mCachedChecksum{d.mCachedChecksum.load()},
  { /* other work to copy Data */ }

  int GetCheckSum() const {
    if (!mCached) {
      mCachedChecksum = CalculateChecksum();
      mCached = true;
    }
    return mCachedChecksum;
  }
  void AddMore(Data const& d) {
    mCached = false;
    // Modify the Data
  }
private:
  int CalculateChecksum() const;
  mutable std::atomic<bool> mCached{false};
  mutable std::atomic<int> mCachedChecksum{0};
};
```



The Importance of Being `const`

The Importance of Being `const`

- Does using `const` generate faster code?

The Importance of Being `const`

- Does using `const` generate faster code?
- Generally no

The Importance of Being `const`

- Does using `const` generate faster code?
- Generally no
- “…when it comes to optimization, `const` is still principally useful as a tool that lets human class designers better implement handcrafted optimizations and less so as a tag for omniscient compilers to automatically generate better code.”

- Herb Sutter, Exceptional C++ Style, Item 24

The Importance of Being `const`

```
void log_it(int);
void modify_it(int&);

void foo(int const& a, int& b)
{
  log_it(a);
  modify_it(b);
  log_it(a);
}
```

The Importance of Being `const`

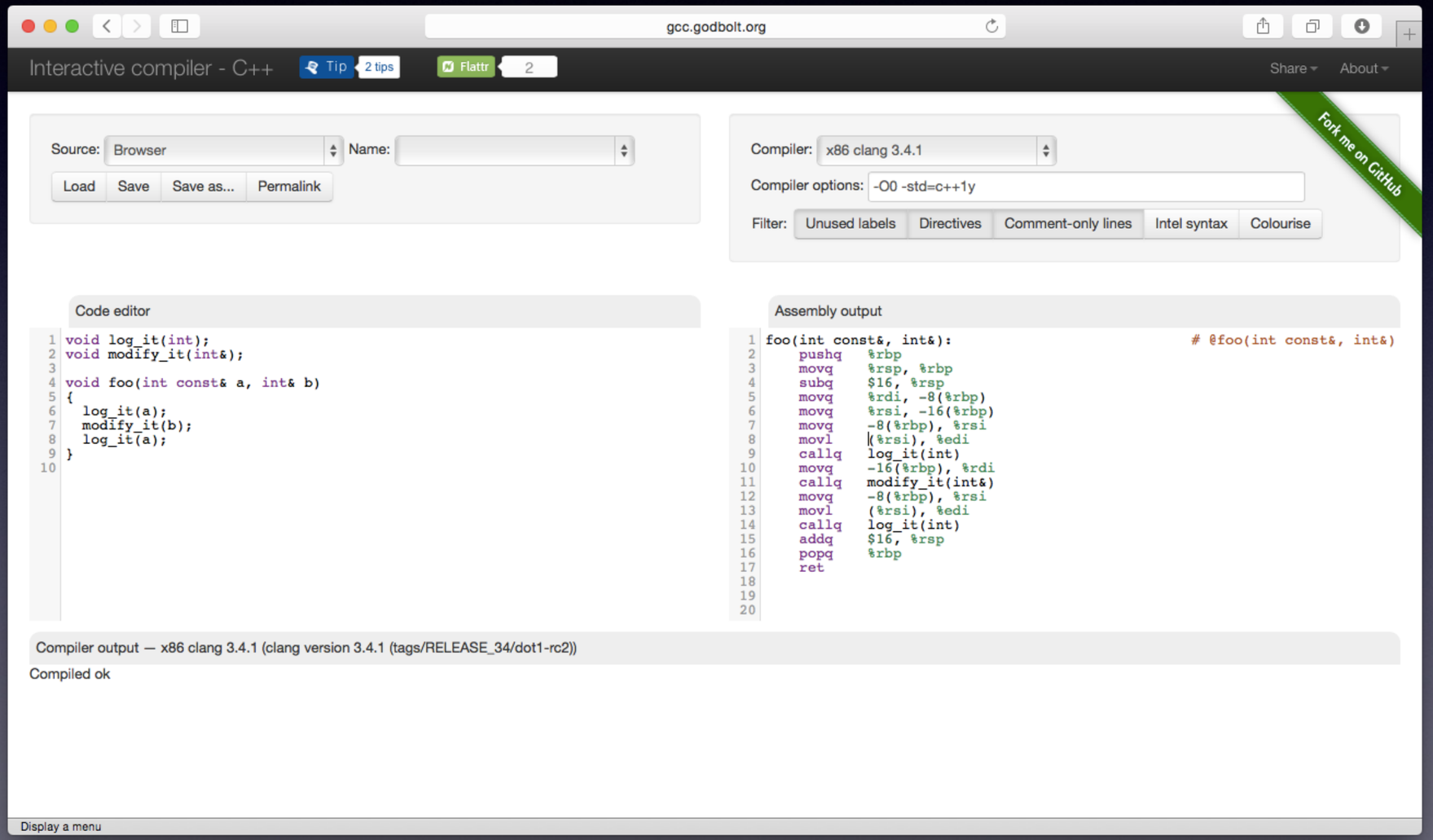
compiled with -O0

```
void log_it(int);
void modify_it(int&);

void foo(int const& a, int& b)
{
  log_it(a);
  modify_it(b);
  log_it(a);
}
```

```
__Z3fooRiRKi:
...
  movl(%rsi), %edi            ## load a
  callq  log_it(int)
  movq-16(%rbp), %rdi
  callq  modify_it(int&)
  movq-8(%rbp), %rsi
  movl(%rsi), %edi            ## load a
  callq  log_it(int)
...
  retq
```

ProTip!

gcc.godbolt.org



The Importance of Being `const`

compiled with -O0

```
void log_it(int);
void modify_it(int&);

void foo(int const& a, int& b)
{
  log_it(a);
  modify_it(b);
  log_it(a);
}
```

```
__Z3fooRiRKi:
...
  movl(%rsi), %edi            ## load a
  callq  log_it(int)
  movq-16(%rbp), %rdi
  callq  modify_it(int&)
  movq-8(%rbp), %rsi
  movl(%rsi), %edi            ## load a
  callq  log_it(int)
...
  retq
```

The Importance of Being `const`

compiled with -O3

```
void log_it(int);
void modify_it(int&);

void foo(int const& a, int& b)
{
  log_it(a);
  modify_it(b);
  log_it(a);
}
```

```
__Z3fooRiRKi:
...
  movl(%rbx), %edi              ## load a
  callq  log_it(int)
  movq%r14, %rdi
  callq  modify_it(int&)
  movl(%rbx), %edi              ## load a
  addq$8, %rsp
  popq%rbx
  popq%r14
  jmp  log_it(int)              ## TAILCALL
```

The compiler always generates two independent memory loads. Why?

The Importance of Being `const`

Consider this case:

```
void log_it(int);
void modify_it(int&);

void foo(int const& a, int& b)
{
  log_it(a);
  modify_it(b);
  log_it(a);
}
```

```
void bar()
{
  int a = 1;
  foo(a, a);
}
```

Compilers cannot make assumptions about what happens at function calls

Review

- `const` helps you find bugs at compile time
- `const` makes it easier to reason about your code
- It helps preserve your class invariants
- Attempt to use `const` following type style to help readability

```
T const t;
```

Review

- `const` member functions should be observably const and thread-safe
- `const` doesn't necessarily make faster code, but it makes more correct code

The Importance of Being `const`

- Questions?