Free Your Functions!

Klaus Iglberger, CppCon 2017, September 29th

Encapsulation

Abstraction / Polymorphism

Cohesion

Flexibility / Extensibility

Reuse / Generality

Testability

Performance

Hypothesis:

Prefer non-member, non-friend functions

The SOLID Principles

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Encapsulation

```
class WebBrowser
{
 public:
   void clearCache();
   void clearHistory();
   void removeCookies();

   void clearEverything()
   {
      clearCache();
      clearHistory();
      removeCookies();
   }
 private:
   // ... the state
};
```

```
class WebBrowser
{
 public:
   void clearCache();
   void clearHistory();
   void removeCookies();
 private:
   // ... the state
};

void clearEverything( WebBrowser& wb )
{
   wb.clearCache();
   wb.clearHistory();
   wb.removeCookies();
}
```

”Object-oriented principles dictate that data and the functions that operate on them should be bundled together, and that suggests that the member function is the better choice. Unfortunately, this suggestion is incorrect. It’s based on a misunderstanding of what being object-oriented means. Object-oriented principles dictate that data should be as encapsulated *as possible. Counterintuitively, the member function [...] actually yields less encapsulation than the non-member [...].”*

(Scott Meyers, Effective C++, 3rd edition)

Coupling

```
class X
{
 public:

   ...




 private:






   std::vector<int> values_;
};
```

```cpp
class X
{
 public:
   void doSomething( ... ) {
      ...
      // Reset values
      ...
   }
   ...
 private:




   std::vector<int> values_;
};
```

```cpp
class X
{
 public:
   void doSomething( ... ) {
      ...
      resetValues();
      ...
   }
   ...
 private:
   void resetValues() {
      for( int& value : values_ )
         value = 0;
   }

   std::vector<int> values_;
};
```

```cpp
class X
{
 public:
   void doSomething( ... ) {
      ...
      resetValues( values_ );
      ...
   }
   ...
 private:




   std::vector<int> values_;
};

void resetValues( std::vector<int>& vec )
{
   for( int& value : vec )
      value = 0;
}
```

Cohesion

(SRP)

```
class X
{
 public:
   ...
 private:
   void resetValues() {
      for( int& value : values_ )
         value = 0;
   }

   std::vector<int> values_;
};
```

```cpp
class X
{
 public:
   ...
 private:




   std::vector<int> values_;
};

void resetValues( std::vector<int>& vec )
{
   for( int& value : vec )
      value = 0;
}
```

Flexibility & Extensibility

(OCP)

```cpp
class X
{
 public:
   ...
 private:






   std::vector<int> values_;
};

void resetValues( std::vector<int>& vec )
{
   for( int& value : vec )
      value = 0;
}
```

Reuse

(DRY)

```cpp
class X
{
    ...
};

class Y
{
 public:
   // ...
 private:
   std::vector<int> indices_;
};

void resetValues( std::vector<int>& vec )
{
   for( int& value : vec )
      value = 0;
}
```

```cpp
class X
{
   ...
};

class Y
{
 public:
   // ...
 private:
   std::vector<int> indices_;
};

void reset( std::vector<int>& vec )
{
   for( int& value : vec )
      value = 0;
}
```

Overloading

(Polymorphism)

```cpp
class X
{
   ...
};

class Y
{
   ...
};
```

```cpp
void reset( std::vector<int>& vec )
{
   for( int& value : vec )
      value = 0;
}
```

```cpp
class X
{
   ...
};

class Y
{
   ...
};

void reset( int& i )
{
   i = 0;
}

void reset( std::vector<int>& vec )
{
   for( int& value : vec )
      reset( value );
}
```

Generic Programming

```cpp
class X
{
   ...
};

class Y
{
   ...
};

void reset( int& i )
{
   i = 0;
}

template< typename T >
void reset( std::vector<T>& vec )
{
   for( T& value : vec )
      reset( value );
}
```

Abstraction

```cpp
class X
{
   ...
};

class Y
{
   ...
};

void reset( int& i )
{
   i = 0;
}

template< typename T >
void reset( std::vector<T>& vec )
{
   for( T& value : vec )
      reset( value );
}
```

Testability

```cpp
class X
{
 public:
   ...
 private:
   void reset() {
      for( int& value : values_ )
         value = 0;
   }

   std::vector<int> values_;


};
```

```cpp
class X
{
 public:
   ...
 private:
   void reset() {
      for( int& value : values_ )
         value = 0;
   }

   std::vector<int> values_;

   friend int testReset(...);
};
```

```cpp
#define private public

class X
{
 public:
   ...
 private:
   void reset() {
      for( int& value : values_ )
         value = 0;
   }

   std::vector<int> values_;


};
```

```cpp
class X
{
 public:
   ...
 private:
   std::vector<int> values_;
};

template< typename T >
void reset( std::vector<T>& vec )
{
   for( T& value : vec )
      reset( value );
}
```

Performance

```cpp
struct S {
   float x, y, z;
   double delta;

   double compute();
};
```

```cpp
double f() {
   S s;
   s.x = /* expensive compute */;
   s.y = /* expensive compute */;
   s.z = /* expensive compute */;
   s.delta = s.x - s.y - s.z;
   return s.compute();
};
```

```
struct S {
   float x, y, z;
   double delta;
};

double compute( S s );
```

```
double f() {
   S s;
   s.x = /* expensive compute */;
   s.y = /* expensive compute */;
   s.z = /* expensive compute */;
   s.delta = s.x - s.y - s.z;
   return compute( s );
};
```

Encapsulation

Cohesion (SRP)

Flexibility / Extensibility (OCP)

Reuse (DRY)

Overloading (Polymorphism)

Generic Programming

Abstraction

Testability

Performance

“Is this even a real idea? Is it used anywhere?”

std::begin, std::cbegin

Defined in header <iterator>

(1)

```
template< class C >
auto begin( C& c ) -> decltype(c.begin());
```

(since C++11) (until C++17)

```
template< class C >
constexpr auto begin( C& c ) -> decltype(c.begin());
```

(since C++17)

(1)

```
template< class C >
auto begin( const C& c ) -> decltype(c.begin());
```

(since C++11) (until C++17)

```
template< class C >
constexpr auto begin( const C& c ) -> decltype(c.begin());
```

(since C++17)

(2)

```
template< class T, std::size_t N >
T* begin( T (&array)[N] );
```

(since C++11) (until C++14)

```
template< class T, std::size_t N >
constexpr T* begin( T (&array)[N] ) noexcept;
```

(since C++14)

(3)

```
template< class C >
constexpr auto cbegin( const C& c ) noexcept(/* see below */)
    -> decltype(std::begin(c));
```

(since C++14)

std::real(std::complex)

Defined in header <complex>

```cpp
template< class T >
T real( const complex<T>& z );
```
(1) (until C++14)

```cpp
template< class T >
constexpr T real( const complex<T>& z );
```
(1) (since C++14)

```cpp
long double real( long double z );
```
(2) (since C++11)

```cpp
template< class DoubleOrInteger >
double real( DoubleOrInteger z );
```
(3) (since C++11)

```cpp
float real( float z );
```
(4) (since C++11)

Returns the real component of the complex number z, i.e. `z.real()`.

“Come on, this is just a reset() function!”

```cpp
class X
{
 public:
   ...
 private:
   std::vector<int> values_;
};

template< typename T >
void reset( std::vector<T> ) {
   for( T& value : values_ )
      reset( value );
}
```

```
class X
{
 public:
   ...
 private:
   std::vector<int> values_;
};

template< typename T >
void reset( std::vector<T> ) {
   ...

}
```

~~“This is just a ‘reset()’ function!”~~

“Functions should be encapsulated!”

```
class X
{
 public:
   void doSomething( ... ) {
      ...
      resetValues();
      ...
   }
   ...
 private:
   void resetValues() {
      for( int& value : values_ )
         value = 0;
   }

   std::vector<int> values_;
};
```

```cpp
class X
{
 public:
   void doSomething( ... ) {
      ...
      resetValues( values_ );
      ...
   }
   ...
 private:




   std::vector<int> values_;
};

void resetValues( std::vector<int>& vec )
{
   for( int& value : vec )
      value = 0;
}
```

```
class X
{
 public:
   void doSomething( ... );







 private:







   std::vector<int> values_;
};
```

~~“Functions should be encapsulated!”~~

“IDEs don’t help with free functions, but with member functions!”

”Note begin(c) and c.begin() for range-for loops and in general code. Why do we/someone have to write both? If c.begin() exists, begin(c) should find it, just as x+y finds the right implementation. … In early 2014, Herb Sutter and I each independently decided to propose a unified syntax. … To my surprise, many people came out strongly against x.f(y) finding f(x,y) – even if member functions were preferred over free-standing functions by the lookup rules. I received email accusing me of “selling out to the OO crowd”.”

(Bjarne Stroustrup)

~~“Programmers cannot find free functions as easily as member functions!”~~

“In combination with ADL free functions mean trouble!”

```cpp
std::complex<double> a, b;

a.min( b );
```

```
src/Sandbox.cpp:517:6: error: no member named 'min' in 'std::__1::complex<float>'
   a.min( b );
   ~ ^
1 error generated.
```

```
std::complex<double> a, b;

min( a, b );
```

```
/opt/local/libexec/llvm-3.9/bin/../include/c++/v1/algorithm:708:71: error: invalid operands to binary expression
      ('const std::__1::complex<float>' and 'const std::__1::complex<float>')
    bool operator()(const _T1& __x, const _T1& __y) const {return __x < __y;}
                                                                  ~~~ ^ ~~~
/opt/local/libexec/llvm-3.9/bin/../include/c++/v1/algorithm:2572:12: note: in instantiation of member function
      'std::__1::__less<std::__1::complex<float>, std::__1::complex<float> >::operator()' requested here
    return __comp(__b, __a) ? __b : __a;
           ^
/opt/local/libexec/llvm-3.9/bin/../include/c++/v1/algorithm:2580:19: note: in instantiation of function template
      specialization 'std::__1::min<std::__1::complex<float>, std::__1::__less<std::__1::complex<float>,
      std::__1::complex<float> > >' requested here
    return _VSTD::min(__a, __b, __less<_Tp>());
                  ^
src/Sandbox.cpp:518:4: note: in instantiation of function template specialization
      'std::__1::min<std::__1::complex<float> >' requested here
   min( a, b );
   ^
1 error generated.
```

~~“In combination with ADL free functions mean trouble!”~~

“I’m using virtual functions, so I cannot use this idea!”

```
class X
{
 public:
   virtual void print( std::ostream& os ) {
      ...
   }
   ...
};

std::ostream& operator<<( std::ostream& os, const X& x )
{
   x.print( os );
}
```

Use free functions in order to …

- … wrap virtual function calls
- … get a homogeneous interface

“Does this mean we should convert to functional programming?”

```cpp
class X
{
   ...
};

class Y
{
   ...
};

void reset( int& i )
{
   i = 0;
}

template< typename T >
void reset( std::vector<T>& vec )
{
   for( T& value : vec )
      reset( value );
}
```

Multiparadigm

“Should we avoid member functions entirely?”

```
if ( f needs to be virtual )
{
   make f a member function of C;
}
else if ( f is operator>> or operator<<  or
          f needs type conversions on its left-most argument )
{
  make f a non-member function;
  if ( f needs access to non-public members of C )
    make f a friend of C;
}
else if ( f can be implemented via C's public interface )
{
  make f a non-member function;
}
else
{
  make f a member function of C;
}
```

(Source: Scott Meyers in C/C++ Users Journal, Feb 2000)

“But I have learned the opposite!”

“Free functions are just backward and C-style programming!”

```
template< typename InputIt, typename OutputIt >
OutputIt copy( InputIt begin, InputIt end
             , OutputIt dst_begin )
{
   ...
}
```

`copy()` adheres to the …

- … Single Responsibility Principle (SRP)
- … Open-Closed Principle (OCP)
- … Interface Segregation Principle (ISP)
- … Dependency Inversion Principle (DIP)

”There was never any question that the [standard template] library represented a breakthrough in efficient and extensible design.”

(Scott Meyers, Effective STL)

~~“Free functions are just backward and C-style programming!”~~

~~Hypothesis:~~

Prefer non-member, non-friend functions

Guideline:

Prefer non-member, non-friend functions

Free Your Functions!

Klaus Iglberger, CppCon 2017, September 29th