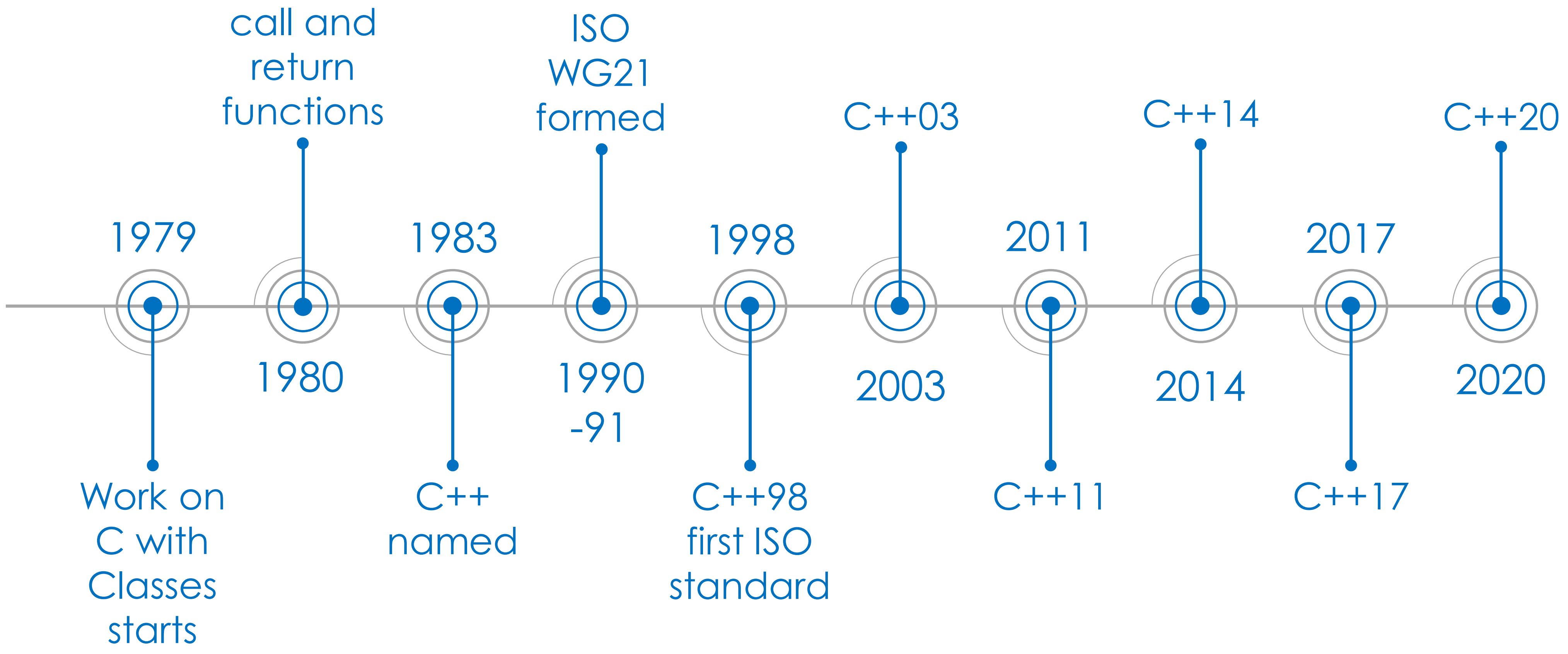
Execute-Around Pointer

Kilian Henneberger

kilis-mail@web.de

C++ Timeline



1979
Work on C with Classes starts
1980
call and return functions
1983
C++ named
1990 -91
ISO WG21 formed
1998
C++98 first ISO standard
2003
C++03
2011
C++11
2014
C++14
2017
C++17
2020
C++20

Motivation

- Wrap calls to an object in pairs of prefix and suffix code
- Simple, reusable and general solution
- Non-intrusive, applicable to existing classes
- E.g. mutex lock/unlock around member functions in a multithreaded environment

```
auto prefix = [](){};
auto suffix = [](){};
string object;
ExecuteAroundPointer wrapper(&object, prefix, suffix);
wrapper->assign("C++"); //prefix(), object.assign("C++"), suffix()
auto length = wrapper->size(); //prefix(), length = object.size(), suffix()
```

Temporary Object Lifetime

- Temporary objects are destroyed as the last step in evaluating the expression that contains the point where they were created
    - If multiple temporary objects were created, they are destroyed in reverse order
- There are exceptions to that rule
    - e.g. binding the temporary to a const lvalue reference or to an rvalue reference

```cpp
struct Trace {
  Trace() { cout << "ctor\n"; }
  ~Trace() { cout << "dtor\n"; }
  void f() { cout << "f\n"; }
};


int main() {
  cout << "1\n";
  Trace().f();
  cout << "2\n";

}
```

Temporary Object Lifetime

- Temporary objects are destroyed as the last step in evaluating the expression that contains the point where they were created
  - If multiple temporary objects were created, they are destroyed in reverse order
- There are exceptions to that rule
  - e.g. binding the temporary to a const lvalue reference or to an rvalue reference

```
struct Trace {
  Trace() { cout << "ctor\n"; }
  ~Trace() { cout << "dtor\n"; }
  void f() { cout << "f\n"; }
};


int main() {
  cout << "1\n";
  Trace().f();
  cout << "2\n";

}
```

```
1
ctor
f
dtor
2
```

Overloading *operator->*

- If a class overloads *operator->*, the *operator->* is called again on the value that it returns
- This process repeats until a raw pointer is returned
- Finally, built-in semantics are applied to that raw pointer

```
struct Inner {
  string s;
  string* operator->() { return &s; }
};

struct Outer {
  Inner inner;
  Inner& operator->() { return inner; }
};

int main() {
  Outer outer;
  outer->assign("C++");
}
```

Overloading *operator->*

- If a class overloads *operator->*, the *operator->* is called again on the value that it returns
- This process repeats until a raw pointer is returned
- Finally, built-in semantics are applied to that raw pointer

```
struct Inner {
  string s;
  string* operator->() { return &s; }
};

struct Outer {
  Inner inner;
  Inner& operator->() { return inner; }
};

int main() {
  Outer outer;
  outer->assign("C++");
}
```

```
outer.operator->().operator->()->assign("C++")
```

Putting it all together – ExecuteAroundPointer

```
template<class Pointer, class Prefix, class Suffix>
class ExecuteAroundPointer {
  Pointer ptr;
  Prefix p;
  Suffix s;

public:
  ExecuteAroundPointer(Pointer ptr, Prefix p, Suffix s)
    : ptr(ptr), p(p), s(s)
  {}
  CallProxy<Pointer&, Suffix&> operator->() {
      p();
      return CallProxy<Pointer&, Suffix&>(ptr, s);
  }
};
```

- `Pointer` could be a raw pointer, a std::shared_ptr<T> or any other type that overloads *`operator->`*

Putting it all together – CallProxy

```
template<class Pointer, class Suffix>
class CallProxy {
  Pointer ptr;
  Suffix s;

public:
  CallProxy(Pointer ptr, Suffix s)
    : ptr(ptr), s(s)
  {}
  Pointer operator->() { return ptr; }
  ~CallProxy() { s(); }
  CallProxy(const CallProxy&) = delete;
  CallProxy& operator=(const CallProxy&) = delete;
};
```

- Important to delete copy- and move-operations as the suffix should only be called once
- Returning the CallProxy requires C++17 (Mandatory Copy Elision)
- There are pre C++17 solutions, too

Putting it all together – Example

```
auto prefix = [](){};
auto suffix = [](){};
string object;
ExecuteAroundPointer wrapper(&object, prefix, suffix);
wrapper->assign("C++");
```

```
wrapper.operator->()
  prefix()
CallProxy temp
temp.operator->()
string* rawPtrToObject
rawPtrToObject->assign("C++")
temp.~CallProxy()
  suffix()
```

Use Case: Multithreading

```
mutex m;
auto prefix = [&] { m.lock(); };
auto suffix = [&] { m.unlock(); };
string object;
ExecuteAroundPointer wrapper(&object, prefix, suffix);
auto action = [&] {
  for (int i = 0; i != 100; ++i) {
    wrapper->push_back('c');
  }
};
array asyncActions = { async(action), async(action), async(action), async(action) };
for (auto& anAsyncAction : asyncActions) {
  anAsyncAction.wait();
}
cout << object.size(); // 400
```

Conclusion

- Use constructor and destructor of a temporary object to "wrap" a member function call by prefix and suffix code
- Recursive execution of `operator->` to first return a temporary proxy object and afterwards the wrapped object
- Solution is simple, reusable and non-intrusive
- Limitations
  - No access to the called member function, its arguments and its result
  - Member access has to happen via `operator->`

References

- "More C++ Idioms/Execute-Around Pointer", Wikibooks, August 2007
  https://en.wikibooks.org/wiki/More_C++_Idioms/Execute-Around_Pointer

- "Wrapping C++ Member Function Calls", Bjarne Stroustrup, June 2000
  http://www.stroustrup.com/wrapper.pdf