Introduction to C++

Some fundamental principles for experienced programmers

Matthew Barulic

Wheego Technologies

March 15, 2018

Preface

This talk...

- ...expects that you have seen C++ code before, but have not necessarily developed in C++ before.
- ...is intended for experienced (but not expert) programmers in some language. It will tend away from covering basic syntax, logic, and control statements.
- ...will occasionally use Python as a reference for comparison, but you don't need to understand Python to get the main points.
- ...is based on modern C++ techniques and best practices. C++ has been widely used for over 20 years. You will find plenty of C++ code and training resources that do not adhere to these practices.

Today's Topics

1. Types
2. Objects and Lifetimes
3. Mythbusting

Types

Types

C++ typing is strong and static.

- **Strong** - Types are checked for compatibility in assignment and use.
- **Static** - Expressions and variables cannot change type at runtime.

Types

For comparison,
Python typing is strong and dynamic.

See: Duck Typing

Types

Every named entity in C++ has a type

Example

```
int a = 10;
  ^


double sum(list<double>& numbers);
   ^          ^      ^
```

Types

Tips about types:

- Types should enforce semantically meaningful code.
- Type names should be very clear for human readers.
- Consider making types even without unique behavior.

Example

```
using Location3D = std::array<double,3>;
using OrientationRPY = std::array<double,3>;
```

auto

Sometimes, being overly explicit with type names can leave code hard to read.

Example

```
MyVeryLongTypeName a = createMyVeryLongTypeName(...
```

auto

The **auto** keyword

- Introduced in C++11
- Asks the compiler to infer types
- Does **NOT** suddenly make C++ dynamically typed
- Types must be known (if only to the compiler) at compile time

Example

```
MyVeryLongTypeName a = createMyVeryLongTypeName(...
auto b = createMyVeryLongTypeName(...);
auto c; // Does not compile! No type info.
```

auto

auto shows up in a lot of code because it can clean up redundant type specifications from the human reader's point of view while still explicitly conveying type information to the compiler.

Example

```
std::vector<int> numbers = ...

for(std::vector<int>::iterator iter = numbers.begin
  iter != numbers.end();
  iter++)
{ ... }

for(auto iter = numbers.begin();
  iter != numbers.end();
  iter++)
{ ... }
```

Templates

Sometimes, we genuinely don't know the type of an object when we **write** our code.

However, the compiler will know all of the types when we **compile** our code.

Templates

Templated entities...

- Use a placeholder type to be filled at compile time.
- Are duplicated for each unique set of template parameters used.
- Cannot be extended with new instantiations at runtime.

Example

```
template<typename T>
T add(const T& a, const T& b) {
  return a + b;
}
```

Templates

Example

```
template<typename T>
T add(const T& a, const T& b) {
  return a + b;
}

auto a = 10;
auto b = 5;
auto c = 7.0;

add<int>(a,b); // OK, a & b are both int's
add<int>(a,c); // OK, c is cast to int
add<>(a,b); // OK, a & b are both int's
add<>(a,c); // BAD, int or double?
```

Example: Boost.Units

- Types facilitate semantic operations
- Boost.Units is a library for dimensional analysis

Example

```
auto distance = 30.0 * meter;
auto elapsed = 10.0 * second;
std::cout << (distance / elapsed) << "\n";
// Output: 3 m s^-1
```

http://www.boost.org/doc/libs/1_65_0/doc/html/boost_units/Quick_Start.html

Types Recap

- Types facilitate semantically meaningful code.
- Types are enforced at compile time.
- **auto** allows us to have clean code without losing types.
- Templates allow us to share code and behavior across types.

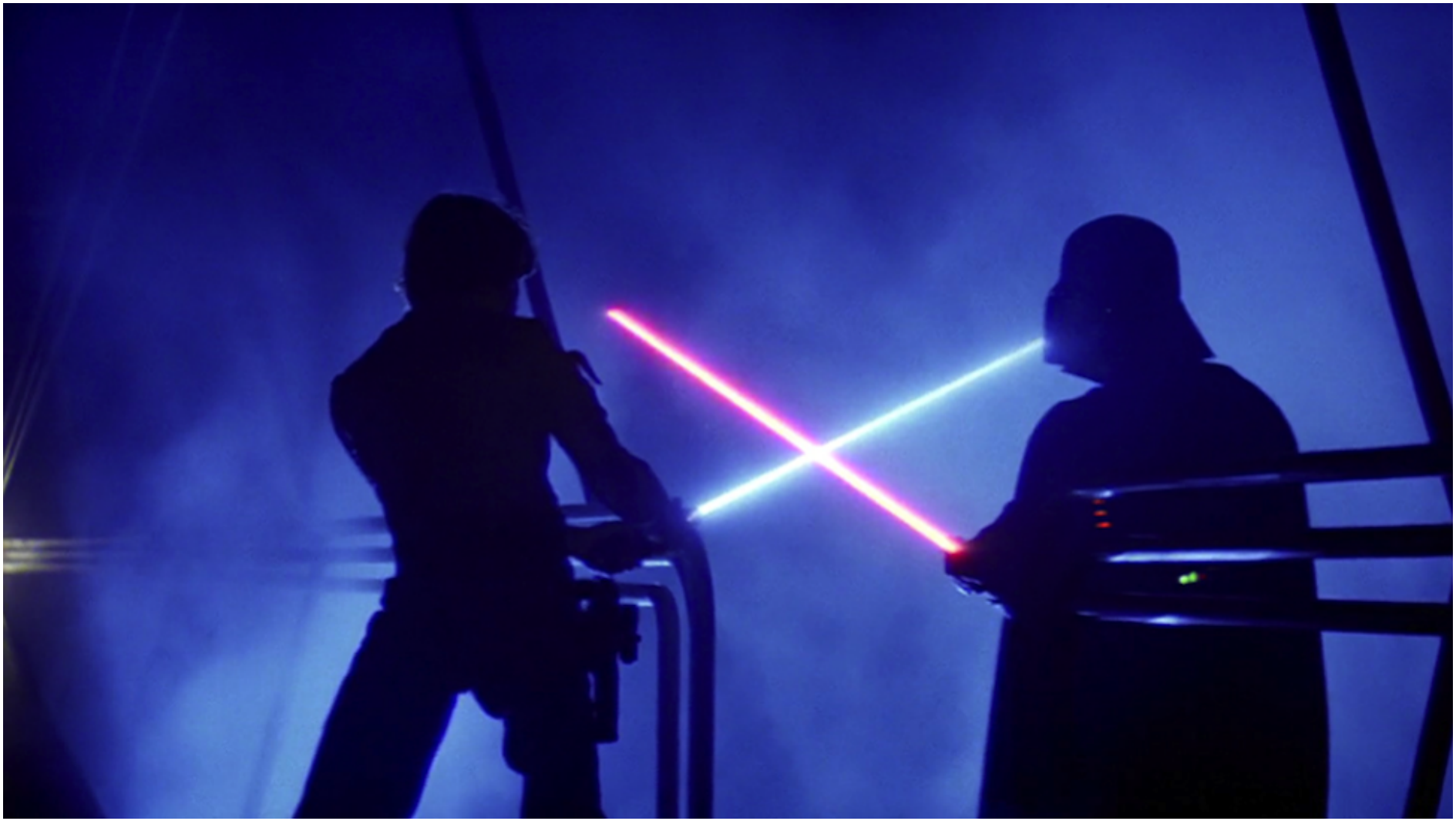
Objects and Lifetimes

Why C++?

Why C++?

Deterministic Object Lifetimes

A brief tangent about lightsabers...



Lifetimes

Object lifetimes...

- Begin after initialization.
- End before destruction.
- Are usually based on variable scope.

Example

```
int main() {
  Robot robot("Robby");
  // ^ Constructor call begins object lifetime
  ...
  return 0;
}
// ^ End of function ends object lifetime
```

Constructors

Construcotrs...

- Initialize an object.
- Are analogous to Python's `__init__()` special functions.

Example

```
class Robot {
  Robot(std::string name)
  { ... }

  Robot()
  : Robot("Unnamed")
  { ... }
};
```

Constructors

Objects cannot be left uninitialized.

Example

```
Robot robot;
// ^ Calls Robot's no-args constructor.
// Compilation fails if Robot doesn't have one.
```

Destructors

Destructors...

- Cleanup any resources owned by the object.
- Are comparable to Python's `__del__()` special functions.

Example

```
class TCPConnection {
  TCPConnection()
  { ... } // Open network socket

  ~TCPConnection()
  { ... } // Close network socket
};
```

RAII

Resource **A**cquisition **I**s **I**nitialization

RAII

- Required resources should be acquired during construction.
- Acquired resources should be released during destruction.
- Eliminates the need for explicit `open()` or `close()` methods.
- Significantly reduces the likelihood of resource mismanagement.

RAII

Example - Python

```
context = zmq.Context()
publisher = ...
...
pulbisher.close()
context.term()
```

Example - C++

```
zmq::Context context;
auto publisher = ...
...
```

Objects and Lifetimes Recap

- C++ maintains clearly defined boundaries on object lifetimes.
- Constructors and destructors give control over either end of an objects lifetime.
- RAII design takes advantage of lifetime mechanics to safely manage resources.

Mythbusting

Myth #1

C++ is complicated.

Myth #1

C++ is complicated.

CONFIRMED

https://www.youtube.com/watch?v=tTexD26jIN4