Modern C++: 11 and up

Neil Moore

Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506

neil@cs.uky.edu

24 January, 2018

Outline

- About C++ and C++ standards
- (Some) new features of C++11 and later
  - Range-based for loops
  - Type inference with `auto`
  - Lambdas
  - `constexpr`
  - Move semantics
- Where to find more information

Guiding principles of C++

- C++ is a "general purpose programming language with a bias towards systems programming" (Stroustrup `http://www.stroustrup.com/C++.html`)
- *Supports* object-oriented programming, doesn't force it.
- "You don't pay for what you don't use"
- "Zero-overhead abstractions"
- This talk will focus on new features of C++11 and up.

C++ standards: history

C++ was developed by Bjarne Stroustrup at AT&T from the late 70s.

- Originally a preprocessor for C, influenced by SIMULA (1960s)

C++ standards: history

C++ was developed by Bjarne Stroustrup at AT&T from the late 70s.

- Originally a preprocessor for C, influenced by SIMULA (1960s)
- 1985: First commercial release, first edition of *The C++ Programming Language* book

C++ standards: history

C++ was developed by Bjarne Stroustrup at AT&T from the late 70s.

- Originally a preprocessor for C, influenced by SIMULA (1960s)
- 1985: First commercial release, first edition of *The C++ Programming Language* book
- 1998: International standardization: ISO/IEC 14882:1998 ("**C++98**")

C++ standards: history

C++ was developed by Bjarne Stroustrup at AT&T from the late 70s.

- Originally a preprocessor for C, influenced by SIMULA (1960s)
- 1985: First commercial release, first edition of *The C++ Programming Language* book
- 1998: International standardization: ISO/IEC 14882:1998 ("**C++98**")
- 2003: Bug fixes: **C++03** (ISO 14882:2003)

C++ standards: history

C++ was developed by Bjarne Stroustrup at AT&T from the late 70s.

- Originally a preprocessor for C, influenced by SIMULA (1960s)
- 1985: First commercial release, first edition of *The C++ Programming Language* book
- 1998: International standardization: ISO/IEC 14882:1998 ("**C++98**")
- 2003: Bug fixes: **C++03** (ISO 14882:2003)
- 2011: Huge revision: **C++11**
  - Took forever: was *C++0x* in development.
  - Still almost entirely backwards-compatible with C++98.
  - "Modern C++" usually means C++11 and up.

C++ standards: history

C++ was developed by Bjarne Stroustrup at AT&T from the late 70s.

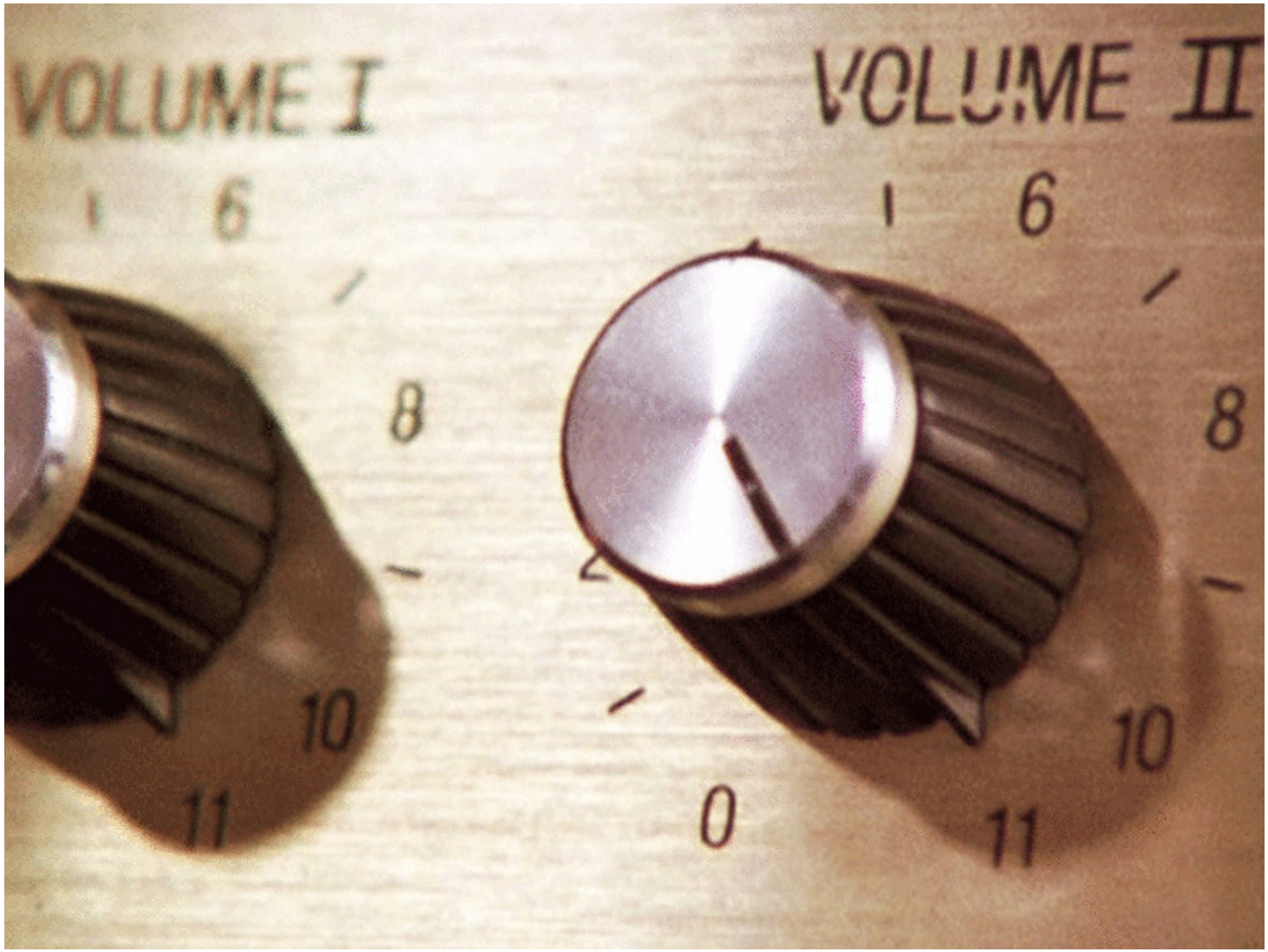
- Originally a preprocessor for C, influenced by SIMULA (1960s)
- 1985: First commercial release, first edition of *The C++ Programming Language* book
- 1998: International standardization: ISO/IEC 14882:1998 ("**C++98**")
- 2003: Bug fixes: **C++03** (ISO 14882:2003)
- 2011: Huge revision: **C++11**
  - Took forever: was *C++0x* in development.
  - Still almost entirely backwards-compatible with C++98.
  - "Modern C++" usually means C++11 and up.
- 2014: Era of clockwork revisions: **C++14**

C++ standards: history

C++ was developed by Bjarne Stroustrup at AT&T from the late 70s.

- Originally a preprocessor for C, influenced by SIMULA (1960s)
- 1985: First commercial release, first edition of *The C++ Programming Language* book
- 1998: International standardization: ISO/IEC 14882:1998 ("**C++98**")
- 2003: Bug fixes: **C++03** (ISO 14882:2003)
- 2011: Huge revision: **C++11**
  - Took forever: was *C++0x* in development.
  - Still almost entirely backwards-compatible with C++98.
  - "Modern C++" usually means C++11 and up.
- 2014: Era of clockwork revisions: **C++14**
- 2017: Tock: **C++17**

C++11



C++ goes to eleven.

Range-based for loops

Above: range-based six loops

Range-based for loops

Range-based `for` loops are syntactic sugar for a common iterator idiom.

- In C++03:

```
const std::set<City> cities = . . . ;
for (std::set<City>::const_iterator i = cities.begin());
     i != cities.end();
     i++)
{
    const City &c = *i;
    . . .
}
```

Range-based for loops

Range-based `for` loops are syntactic sugar for a common iterator idiom.

- In C++03:

```
const std::set<City> cities = . . . ;
for (std::set<City>::const_iterator i = cities.begin());
     i != cities.end();
     i++)
{
    const City &c = *i;
    . . .
}
```

- In C++11:

```
const std::set<City> cities = . . . ;
for (const City &c : cities)
{
    . . .
}
```

Range-based for loops

- Range for with plain arrays, not just data structures with iterators. In C++11:

```
char name[] = "Stroustrup";
for (char letter : name)
    cout << letter << '\n';
```

Range-based for loops

- Range for with plain arrays, not just data structures with iterators. In C++11:

```
char name[] = "Stroustrup";
for (char letter : name)
    cout << letter << '\n';
```

- Loop variable can be a plain variable, reference, const reference, ...:

```
char name[] = "Stroustrup";
for (char &letter : name)
    letter = '*';
```

Range-based for loops

- Range for with plain arrays, not just data structures with iterators. In C++11:

```
char name[] = "Stroustrup";
for (char letter : name)
    cout << letter << '\n';
```

- Loop variable can be a plain variable, reference, const reference, ...:

```
char name[] = "Stroustrup";
for (char &letter : name)
    letter = '*';
```

- No access to the actual iterator!
  - If you need the index (0, 1, 2, ...), don't use range for.

Type inference with auto

But does it support type inference?

Type inference with auto

C++11 can in many instances infer types of variables automatically:

```
std::vector<Animal> pets = get_pets();
auto opal = pets[2];  // copy: Animal opal = pets[2];
auto &gus = pets[3];  // ref:  Animal &gus = pets[3];
```

Type inference with auto

C++11 can in many instances infer types of variables automatically:

```
std::vector<Animal> pets = get_pets();
auto opal = pets[2];  // copy: Animal opal = pets[2];
auto &gus = pets[3];  // ref:  Animal &gus = pets[3];

// std::vector<Animal>::iterator start = . . .
auto start = pets.begin();
```

Type inference with auto

C++11 can in many instances infer types of variables automatically:

```
std::vector<Animal> pets = get_pets();
auto opal = pets[2];  // copy: Animal opal = pets[2];
auto &gus = pets[3];  // ref:  Animal &gus = pets[3];

// std::vector<Animal>::iterator start = . . .
auto start = pets.begin();
```

C++14 made auto return types work for most functions:

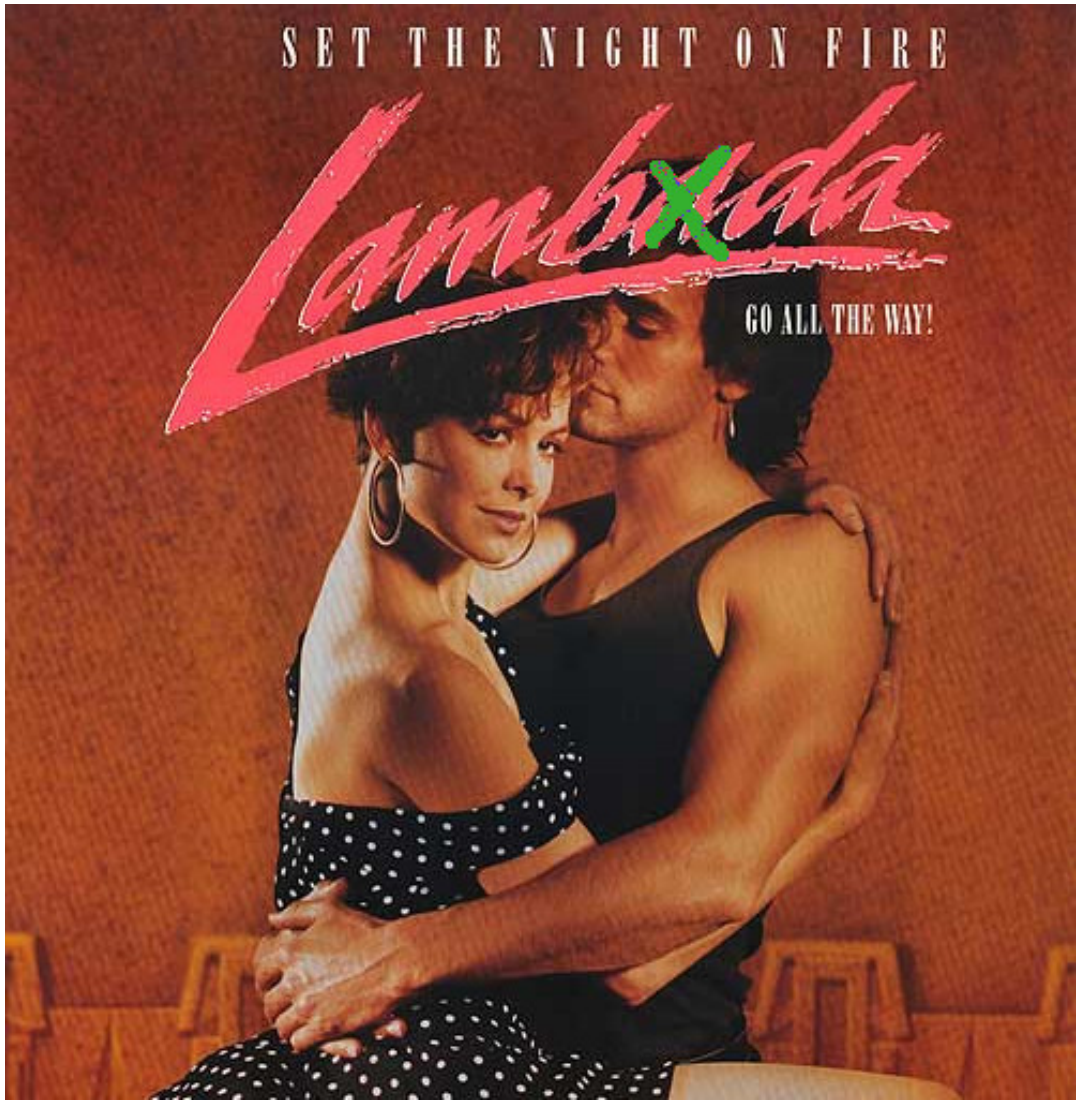
```
auto myfn(bool large)   // double myfn(bool large)
{
  if (large)
      return std::exp(20.0);
  else
      return 20.0;
}
```

Type inference with auto, cont'd

AAA style: "almost always auto":

```
auto num_users  = 5;
auto name_array = "hello world";
auto name_str   = std::string{"hello world"};
auto name_str2  = "hello world"s; // Same, since C++14
```

Lambdas



Can you believe there were *two* movies in 1990 about the lambada dance?
Both were box-office flops.

Lambdas

Lambdas are unnamed functions that can *capture* variables in the surrounding scope:

```
int min_value = flag ? 3 : 5;
auto it = std::find_if(vec.begin(), vec.end(),
                       [min_value](int num) -> bool {
                           return num >= min_value
                       });
```

- `[captures] (arguments) -> ret_type { body }`

Lambdas

Lambdas are unnamed functions that can *capture* variables in the surrounding scope:

```
int min_value = flag ? 3 : 5;
auto it = std::find_if(vec.begin(), vec.end(),
                       [min_value](int num) -> bool {
                           return num >= min_value
                       });
```

- `[captures] (arguments) -> ret_type { body }`
- Square brackets: variables to capture
  - `&var` to capture by reference
  - `=` for all variables used
  - `&` for all variables used, by reference.
- Parentheses: function parameters
- `->`: return type

constexpr

Apologies to Allie Brosh of Hyperbole and a Half.

`constexpr`

The keyword `constexpr` tells the compiler that a variable or function can and should be computed at compile time.

- Faster run-time, slower compilation.

constexpr

The keyword `constexpr` tells the compiler that a variable or function can and should be computed at compile time.

- Faster run-time, slower compilation.
- C++11 only allowed constexpr for very simple functions.
  - C++14 greatly expanded what can be done at compile time.

```
// OK in C++11, but a loop or if would only work in C++14:
constexpr long long factorial(int n)
{
  return n <= 0 ? 1
                : n * factorial(n - 1);
}
```

constexpr

The keyword `constexpr` tells the compiler that a variable or function can and should be computed at compile time.

- Faster run-time, slower compilation.
- C++11 only allowed constexpr for very simple functions.
  - C++14 greatly expanded what can be done at compile time.

```
// OK in C++11, but a loop or if would only work in C++14:
constexpr long long factorial(int n)
{
  return n <= 0 ? 1
                : n * factorial(n - 1);
}

// Computed at compile-time:
constexpr auto thirteen_fact = factorial(13);
```

constexpr

The keyword `constexpr` tells the compiler that a variable or function can and should be computed at compile time.

- Faster run-time, slower compilation.
- C++11 only allowed constexpr for very simple functions.
  - C++14 greatly expanded what can be done at compile time.

```
// OK in C++11, but a loop or if would only work in C++14:
constexpr long long factorial(int n)
{
  return n <= 0 ? 1
                : n * factorial(n - 1);
}

// Computed at compile-time:
constexpr auto thirteen_fact = factorial(13);

int x = . . .;
auto x_fact = factorial(x); // Computed at run-time
```

constexpr

The keyword `constexpr` tells the compiler that a variable or function can and should be computed at compile time.

- Faster run-time, slower compilation.
- C++11 only allowed constexpr for very simple functions.
  - C++14 greatly expanded what can be done at compile time.

```
// OK in C++11, but a loop or if would only work in C++14:
constexpr long long factorial(int n)
{
  return n <= 0 ? 1
                : n * factorial(n - 1);
}

// Computed at compile-time:
constexpr auto thirteen_fact = factorial(13);

int x = . . .;
auto x_fact = factorial(x); // Computed at run-time

constexpr auto x_fact_2 = factorial(x); // ERROR
```

Move semantics

Imagine this truck is full of semantics.

Move semantics

One of the most significant changes in C++11 was the addition of *move constructors*, *move assignment operators*, and *rvalue references*.

- Consider a function that takes a (potentially long) string argument and modifies it internally.

Move semantics

One of the most significant changes in C++11 was the addition of *move constructors*, *move assignment operators*, and *rvalue references*.

- Consider a function that takes a (potentially long) string argument and modifies it internally.
- C++98 and 03 options:
  - Call-by-value: Requires a copy: expensive!
  - Call-by-reference: Doesn't work for temporaries.
  - Call-by-const-reference: Can't modify the argument.

Move semantics

One of the most significant changes in C++11 was the addition of *move constructors*, *move assignment operators*, and *rvalue references*.

- Consider a function that takes a (potentially long) string argument and modifies it internally.
- C++98 and 03 options:
  - Call-by-value: Requires a copy: expensive!
  - Call-by-reference: Doesn't work for temporaries.
  - Call-by-const-reference: Can't modify the argument.
- *Move semantics*: move data directly from temporary into argument.
  - Usually just a pointer adjustment: cheap!
  - The data is lost from the original location.
  - But that's a temporary, so no problem!
  - "Valid but unspecified state".

Move semantics

One of the most significant changes in C++11 was the addition of *move constructors*, *move assignment operators*, and *rvalue references*.

- Consider a function that takes a (potentially long) string argument and modifies it internally.
- C++98 and 03 options:
  - Call-by-value: Requires a copy: expensive!
  - Call-by-reference: Doesn't work for temporaries.
  - Call-by-const-reference: Can't modify the argument.
- *Move semantics*: move data directly from temporary into argument.
  - Usually just a pointer adjustment: cheap!
  - The data is lost from the original location.
  - But that's a temporary, so no problem!
  - "Valid but unspecified state".
- Most of the standard data structures support move semantics.

Move semantics, cont'd

To add move semantics to your own data structures, add a move constructor and a move assignment operator with an *rvalue reference* parameter (&&):

```
class my_string {
  char *data;
  std::size_t length;
public:
```

Move semantics, cont'd

To add move semantics to your own data structures, add a move constructor and a move assignment operator with an *rvalue reference* parameter (&&):

```
class my_string {
  char *data;
  std::size_t length;
public:

  my_string(my_string &&other)
      : data(other.data), length(other.length)
  { other.data = nullptr; other.length = 0; }
```

Move semantics, cont'd

To add move semantics to your own data structures, add a move constructor and a move assignment operator with an *rvalue reference* parameter (&&):

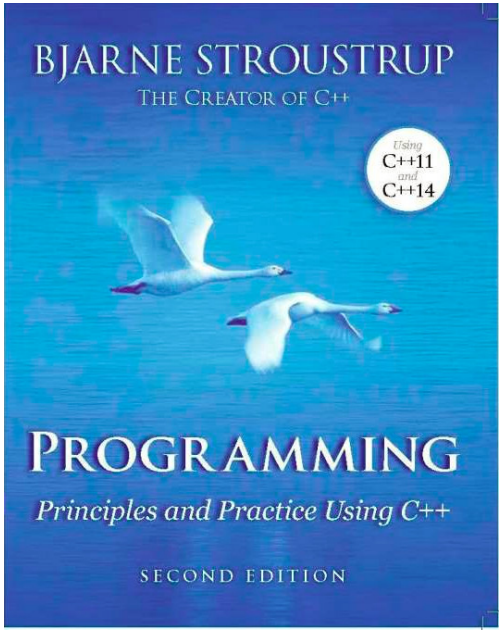
```
class my_string {
  char *data;
  std::size_t length;
public:

  my_string(my_string &&other)
      : data(other.data), length(other.length)
  { other.data = nullptr; other.length = 0; }

  my_string &operator =(my_string &&other)
  { std::swap(data, other.data);
    std::swap(length, other.length);
  }
```

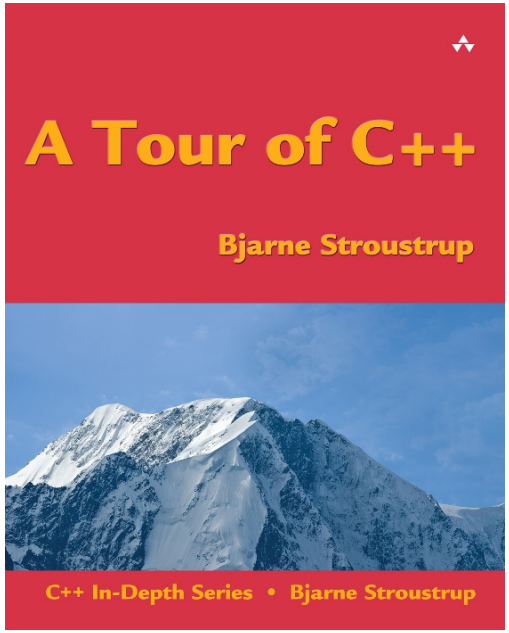
Move semantics, cont'd

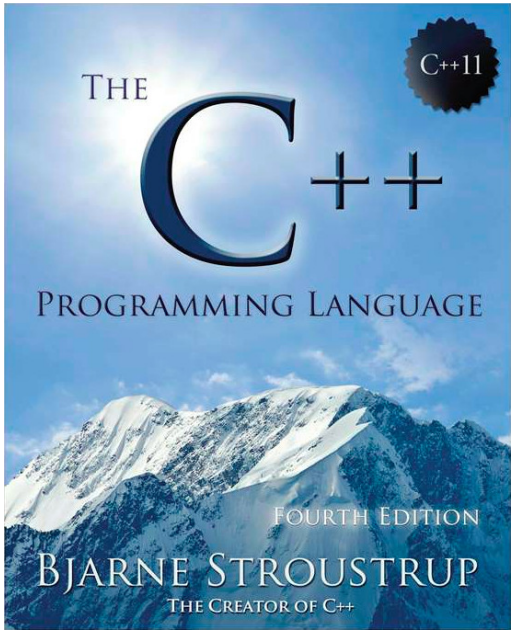
To add move semantics to your own data structures, add a move constructor and a move assignment operator with an *rvalue reference* parameter (&&):

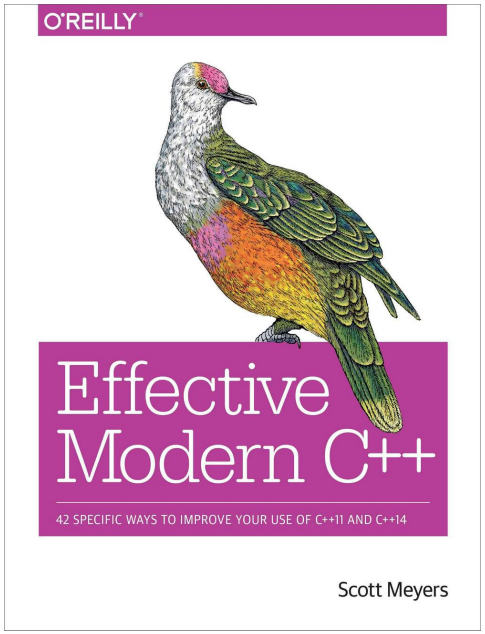
```
class my_string {
  char *data;
  std::size_t length;
public:

  my_string(my_string &&other)
      : data(other.data), length(other.length)
  { other.data = nullptr; other.length = 0; }

  my_string &operator =(my_string &&other)
  { std::swap(data, other.data);
    std::swap(length, other.length);
  }

  // Probably also need a copy constructor and assignment operator
  // (const my_string &other)
}
```

Further reading









Books

- Stroustrup, *Programming: Principles and Practice Using C++*, 2nd ed. 2014.
  - Introductory programming textbook using modern C++.
- Stroustrup, *A Tour of C++*. 2013.
  - Overview of modern C++ for people who know old-fashioned C++.
  - 2nd ed. will cover C++20, but that's a few years off.
- Stroustrup, *The C++ Programming Language*, 4th ed. 2013.
  - The complete reference manual for C++. Covers up to C++11.
- Scott Meyers, *Effective Modern C++*
  - Using new features of C++11 and C++14 to maximum effect.

Free online material

- CppCon videos: `https://www.youtube.com/user/CppCon/videos`
  - Annual C++ conference.
  - Lots of excellent talks, from beginner to expert.
  - Great (IMHO) speakers: Scott Meyers, Herb Sutter, Chandler Carruth, Stephan T. Lavavej, Bjarne Stroustrup

Free online material

- CppCon videos: `https://www.youtube.com/user/CppCon/videos`
  - Annual C++ conference.
  - Lots of excellent talks, from beginner to expert.
  - Great (IMHO) speakers: Scott Meyers, Herb Sutter, Chandler Carruth, Stephan T. Lavavej, Bjarne Stroustrup
- cppreference: `http://en.cppreference.com/`
  - Excellent, but very technical, reference manual for C++.
  - Indicates changes between C++03, C++11, C++14, C++17.
  - Almost every page has an example!

Free online material

- CppCon videos: `https://www.youtube.com/user/CppCon/videos`
  - Annual C++ conference.
  - Lots of excellent talks, from beginner to expert.
  - Great (IMHO) speakers: Scott Meyers, Herb Sutter, Chandler Carruth, Stephan T. Lavavej, Bjarne Stroustrup
- cppreference: `http://en.cppreference.com/`
  - Excellent, but very technical, reference manual for C++.
  - Indicates changes between C++03, C++11, C++14, C++17.
  - Almost every page has an example!
- C++17 working draft: `https://isocpp.org/std/the-standard`
  - The official standard costs money, but the last working draft before the ballot is available for free.
  - Hope you like standardese!
  - Mostly useful for C++ implementors...

Free online material

- CppCon videos: `https://www.youtube.com/user/CppCon/videos`
  - Annual C++ conference.
  - Lots of excellent talks, from beginner to expert.
  - Great (IMHO) speakers: Scott Meyers, Herb Sutter, Chandler Carruth, Stephan T. Lavavej, Bjarne Stroustrup
- cppreference: `http://en.cppreference.com/`
  - Excellent, but very technical, reference manual for C++.
  - Indicates changes between C++03, C++11, C++14, C++17.
  - Almost every page has an example!
- C++17 working draft: `https://isocpp.org/std/the-standard`
  - The official standard costs money, but the last working draft before the ballot is available for free.
  - Hope you like standardese!
  - Mostly useful for C++ implementors...

Thank you! Questions or remarks?