# C++11

## Features and Tricks

Daniel Taliun

July 28, 2016

# From C++98 to C++11

- C++11 introduces >40 new features to C++ language
- Most recent versions of wide-spread compilers (GCC, Clang, MS Visual C++) support C++11 completely

en.cppreference.com/w/cpp/compiler_support

# Roadmap

1. Range-for loop
2. auto
3. nullptr
4. Lambda expressions
5. Smart pointers

# Range-for loop

```
int array[] = {1, 2, 3, 4, 5}; // initializer-list!

for (int a : array) {
        std::cout << a << std::endl;
}
```

# Range-for loop

```
int array[] = {1, 2, 3, 4, 5}; // initializer-list!

for (int a : array) {
      std::cout << a << std::endl;
}
```

```
int array[] = {1, 2, 3, 4, 5}; // initializer-list!

for (int& a : array) {
      std::cout << a << std::endl;
}
```

```
int array[] = {1, 2, 3, 4, 5}; // initializer-list!

for (const int& a : array) {
      std::cout << a << std::endl;
}
```

# auto

- Forces initialization

```
int x1; // compiles, but x1 value is unknown

auto x2; // compilation error!

auto x3 = 1; // Good, x3 is int and has known value.
```

# auto

- Can save from performance issues

```
std::unordered_map<std::string, int> dict;

for (const std::pair<std::string, int>& entry : dict) {
        // some code
}
```

# auto

- Can save from performance issues

```
std::unordered_map<std::string, int> dict;

for (const std::pair<std::string, int>& entry : dict) {
        // some code
}
```

Copying!
From std::pair<**const** std::string, int> to std::pair<std::string, int>

# auto

- Can save from performance issues

```
std::unordered_map<std::string, int> dict;

for (const auto& entry : dict) {
        // some code
}
```

Good! No copying - correct type is reduced automatically!

# auto

- Do b1 and b2 have the same type?

```
std::vector<bool> v = {true, false};

bool b1 = v[1];
auto b2 = v[1];
```

# auto

- b1 is bool, but b2 is std::vector<bool>::reference
- Using b2 is dangerous – it may contain dangling pointer if v is destroyed!

```
std::vector<bool> v = {true, false};

bool b1 = v[1];
auto b2 = v[1];
```

# auto

- Special treatment in range-for loop

```
std::vector<bool> v = {true, false};

for (auto& a : v) {
        a = true;
}
```

Compilation error!

```
std::vector<bool> v = {true, false};

for (auto&& a : v) {
        a = true;
}

// or if you don't need modify a
for (const auto& a : v) {
        // some code
}
```

Good!

# nullptr

- Try to use nullptr instead of 0 or NULL

```
void f(int);
void f(bool);
void f(void*);

f(0); // calls f(int), not f(void*)

f(NULL); // doesn't compile with GCC 4.9, otherwise typically would call f(int)

f(nullptr); // correctly calls f(void*)
```

# Lambda Expressions

- Anonymous functions that are super useful when working with STL algorithms such as std::find_if, std::remove_if, std::count_if, std::sort, std::nth_element, std::lower_bound and etc.

```
std::vector<int> v = {1, 2, 3, 4, 5};

auto e = std::find_if(v.begin(), v.end(),
                [](int value) { return 2 < value && value < 4; } );

std::cout << *e << std::endl;
```

# Lambda Expressions

- Can be copied using auto

```
int x = 5;

auto c1 = [x](int y) { return x * y; };
auto c2 = c1;

std::cout << c1(2) << std::endl; // 10
std::cout << c2(3) << std::endl; // 15
```

# Lambda Expressions

- Capture by value [x] (or [=] for all variables)

```
int x = 5;

auto c1 = [x](int y) { return x * y; };
auto c2 = c1;

std::cout << c1(2) << std::endl; // 10
std::cout << c2(3) << std::endl; // 15

x = 10;

std::cout << c1(2) << std::endl; // 10
std::cout << c2(3) << std::endl; // 15
```

# Lambda Expressions

- Capture by reference [&x] (or [&] for all variables)

```
int x = 5;

auto c1 = [&x](int y) { return x * y; };
auto c2 = c1;

std::cout << c1(2) << std::endl; // 10
std::cout << c2(3) << std::endl; // 15

x = 10;

std::cout << c1(2) << std::endl; // 20
std::cout << c2(3) << std::endl; // 30
```

# Lambda Expressions

- You can store them in std::function objects

```
std::vector<std::function<int(int)>> functions;

functions.emplace_back([](int y) { return 5 * y; } );

std::cout << functions[0](2) << std::endl; // 10
```

# Lambda Expressions

- Be careful with reference capture! Watch for dangling references!

```
std::vector<std::function<int(int)>> functions;

{
        auto x = 5;
        functions.emplace_back([&](int y) { return x * y; } );
}

std::cout << functions[0](2) << std::endl; // Unknown! x might be destroyed...
```

# Smart Pointers

- std::auto_ptr (Deprecated)
- std::unique_ptr
- std::shared_ptr
- std::weak_ptr

# std::unique_ptr

- For exclusive ownership resource management
- Moving is ok but copying is not allowed
- On destruction it destroys its resource

```
// open new scope
{
	std::unique_ptr<int[]> array(new int[10]);

	for (int i = 0; i < 10; ++i) { // work with array as usual
		array[i] = i;
	}
}
// close the scope
// array is automatically destroyed by unique_ptr (delete[] is called)
```

# std::unique_ptr

- What about C-style array?

```
// open new scope
{
        std::unique_ptr<int> array((int*)malloc(10 * sizeof(int)));
        for (int i = 0; i < 10; ++i) {
                array.get()[i] = i;
        }
}
// close the scope
```

# std::unique_ptr

- What about C-style array?

Bad! unique_ptr calls delete, but we need free()!

```
// open new scope
{
	std::unique_ptr<int> array((int*)malloc(10 * sizeof(int)));
	for (int i = 0; i < 10; ++i) {
		array.get()[i] = i;
	}
}
// close the scope
```

# std::unique_ptr

- What about C-style array?

```
// open new scope
{
      auto deleter = [] (int* ptr) {
                        free(ptr);
                        std::cout << "I am free." << std::endl;
                  };
      std::unique_ptr<int, decltype(deleter)> array(
                                    (int*)malloc(10 * sizeof(int)), deleter);
      for (int i = 0; i < 10; ++i) {
            array.get()[i] = i;
      }
}
// close the scope
// Excellent!!! You should see "I am free." message!
```

# std::unique_ptr

- What about pointer size?

```
std::unique_ptr<int[]> array1(new int[10]);
std::unique_ptr<int, decltype(deleter)> array2(
                         (int*)malloc(10 * sizeof(int)), deleter);
int* array3 = new int[10];

std::cout << sizeof(array1) << std::endl; // 8
std::cout << sizeof(array2) << std::endl; // 8
std::cout << sizeof(array3) << std::endl; // 8
```

# std::unique_ptr

- What about pointer size? Avoid non-lambda deleters!

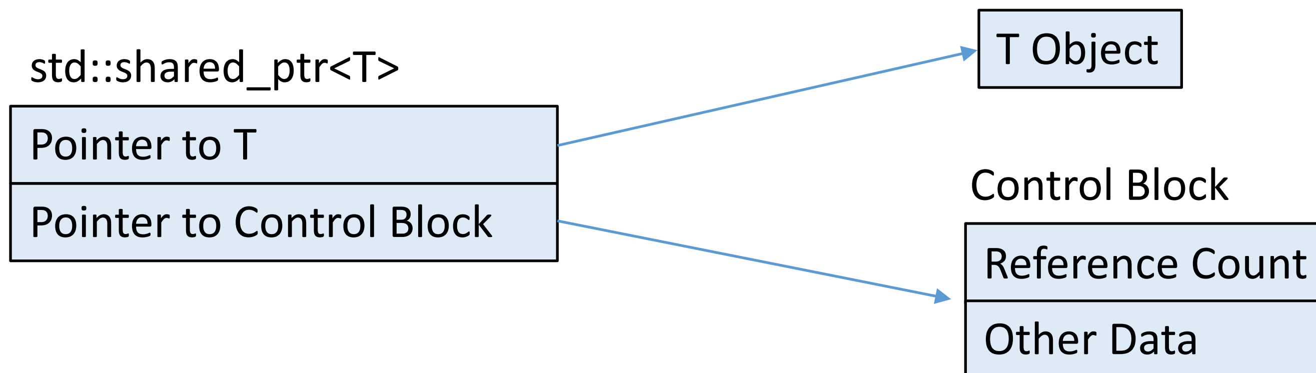
```
void deleter2(int* ptr) {
      free(ptr);
      std::cout << "I am free." << std::endl;
}

std::unique_ptr<int[]> array1(new int[10]);
std::unique_ptr<int, decltype(deleter)> array2(
                                 (int*)malloc(10 * sizeof(int)), deleter);
std::unique_ptr<int, void(*)(int*)> array3(
                                 (int*)malloc(10 * sizeof(int)), deleter2);

std::cout << sizeof(array1) << std::endl; // 8
std::cout << sizeof(array2) << std::endl; // 8
std::cout << sizeof(array3) << std::endl; // 16
```

# std::shared_ptr

- Counts number of references that point to the resource
- Resource is destroyed when the last shared pointer is destroyed
- Twice as large as a raw pointer
- Increment & decrement of reference count must be atomic
- Doesn’t have support for delete[]



# std::shared_ptr

```
auto deleter = [] (int* ptr) {
        delete[] ptr;
        std::cout << "I am free." << std::endl;
};

{
        std::shared_ptr<int> array1(new int[10], deleter);
        std::cout << array1.use_count() << std::endl; // 1
        {
                std::shared_ptr<int> array2 = array1;
                std::cout << array1.use_count() << std::endl; // 2
                std::cout << array2.use_count() << std::endl; // 2
        }
        // array2 shared_ptr is destroyed, but raw array is not deleted.
        std::cout << array1.use_count() << std::endl; // 1
}
// "I am free."
```

# std::weak_ptr

- Same as std::shared_ptr but doesn’t participate in reference count
- Stores pointers that can dangle
- Can be constructed only from another shared_ptr

# Important Features to Learn

- Move semantics
- Rvalue/Lvalue references
- Regular expressions
- Concurrency API

# Based on