C++11: 10 Features You Should be Using

Gordon Brown

@AerialMantis

R&D Runtime Engineer

Codeplay Software Ltd.

Agenda

- Default and Deleted Methods
- Static Assertions
- Delegated and Inherited Constructors
- Null Pointer Type
- Enum Classes
- Automatic Type Deduction
- Ranged For Loops
- Smart Pointers
- Lambdas and Function Type
- Move Semantics

Default and Deleted Functions

- New 'default' keyword specifies default constructor or operator
- Useful when partially implementing the rule of three (or five)

Default and Deleted Functions

- **Example:**

```
class foo {
public:
    foo();
    foo &operator= (const foo &rhs) = default;
    foo (const foo &rhs) = default;
    ~foo();
};
```

foo uses the default copy constructor and assignment operator

Default and Deleted Functions

- New 'delete' keyword specifies a constructor or operator as unavailable
- Useful for restricting the way a type can be used

Default and Deleted Functions

- **Example:**

```
class foo {
public:
    foo();
    foo &operator= (const foo &rhs) = delete;
    foo (const foo &rhs) = delete;
    ~foo();
};
```

foo is non-copyable

Agenda

- ✔ Default and Deleted Methods
- Static Assertions
- Delegated and Inherited Constructors
- Null Pointer Type
- Enum Classes
- Automatic Type Deduction
- Ranged For Loops
- Smart Pointers
- Lambdas and Function Type
- Move Semantics

Static Assertions

- Compile time assertions
- Useful for generating compiler time errors for templates

Static Assertions

- **Example:**

```
template <typename T, unsigned dim>
class foo {
static_assert(dim > 0, "foo must be 1 dimensions or greater");
public:
    foo();
};
```

foo<int, 0> f;
would give this compiler error

```
error C2338: foo must be 1 dimensions or greater
```

Agenda

- ✔ Default and Deleted Methods
- ✔ Static Assertions
- Delegated and Inherited Constructors
- Null Pointer Type
- Enum Classes
- Automatic Type Deduction
- Ranged For Loops
- Smart Pointers
- Lambdas and Function Type
- Move Semantics

Delegated and Inherited Constructors

- Delegated constructors allows one constructor to call another
- Useful for avoiding code duplication for initialization

Delegated and Inherited Constructors

- **Example:**

```
class foo {
public:
    foo (int x, int y, int z)
     : m_x(x), m_y(y), m_z(z) {}
    foo ()
     : foo(0, 0, 0) {}
private:
    int m_x, m_y, m_z;
};
```

foo's default constructor calls
foo's second constructor
foo() → foo(int, int, int)

Delegated and Inherited Constructors

- Inherited constructors allows a class to inherit constructors from its base class
- Useful for avoiding constructors that simply pass on the same parameters

Delegated and Inherited Constructors

- **Example:**

```
class foo {
public:
    foo ();
    foo (int x);
};

class bar : public foo {
    using foo::foo;
};
```

bar inherits the constructors:
bar ()
bar (int)

Agenda

- ✔ Default and Deleted Methods
- ✔ Static Assertions
- ✔ Delegated and Inherited Constructors
- Null Pointer Type
- Enum Classes
- Automatic Type Deduction
- Ranged For Loops
- Smart Pointers
- Lambdas and Function Type
- Move Semantics

Null Pointer Type

- New 'nullptr' keyword
- Alias for the 'nullptr_t' type
- Comparable to any pointer
- Not implicitly convertible or comparable to integral types, except bool

Null Pointer Type

- **Example:**

```
void foo (int i);
void foo (char *p);

int main () {

    foo(NULL);
    foo(nullptr);
}
```

Which constructor is called when passing NULL?

Which constructor is called when passing nullptr?

Agenda

- ✔ Default and Deleted Methods
- ✔ Static Assertions
- ✔ Delegated and Inherited Constructors
- ✔ Null Pointer Type
- Enum Classes
- Automatic Type Deduction
- Ranged For Loops
- Smart Pointers
- Lambdas and Function Type
- Move Semantics

Enum Classes

- Improvement on traditional enums
- Allows forward declarations
- Does not pollute top level namespace
- Not implicitly convertible to integers
- Can specify the element size

Enum Classes

- **Example:**

```
enum class animal_type : int {
    dog = 0,
    cat,
    mouse
};
```

Each element of the enum is an integer.

```
animal_type animalType = animal_type::dog
```

Agenda

- ✔ Default and Deleted Methods
- ✔ Static Assertions
- ✔ Delegated and Inherited Constructors
- ✔ Null Pointer Type
- ✔ Enum Classes
- Automatic Type Deduction
- Ranged For Loops
- Smart Pointers
- Lambdas and Function Type
- Move Semantics

Automatic Type Deduction

- New 'auto' keyword allows compile type deduction
- Useful when a type is very complex such as iterators or functions

Automatic Type Deduction

- **Example:**

```
int i = 4;
auto i = 4;
foo f = func();
auto f = func();
std::vector<int>::iterator it = vec.begin();
auto it = vec.begin();
std::function<void(std::vector<int>)> getSize =
  [](std::vector<int> vec) { return vec.size(); };
auto getSize = [](std::vector<int> vec) { return vec.size(); };
```

Agenda

- ✔ Default and Deleted Methods
- ✔ Static Assertions
- ✔ Delegated and Inherited Constructors
- ✔ Null Pointer Type
- ✔ Enum Classes
- ✔ Automatic Type Deduction
- Ranged For Loops
- Smart Pointers
- Lambdas and Function Type
- Move Semantics

Ranged For Loops

- Simpler syntax for iterable types
- Can be used on any type that has either:
  - begin() and end() methods
  - begin(std::vector) and end(std::vector) functions

Ranged For Loops

- **Example:**

```
int sum = 0;
for (std::vector<int>::iterator it = vec.begin();
     it != vec.end(); ++it) {
    sum += *it;
}
int sum = 0;
for (int i : vec) {
    sum += i;
}
```

It is also possible to use auto as the type for i

Agenda

- ✔ Default and Deleted Methods
- ✔ Static Assertions
- ✔ Delegated and Inherited Constructors
- ✔ Null Pointer Type
- ✔ Enum Classes
- ✔ Automatic Type Deduction
- ✔ Ranged For Loops
- Smart Pointers
- Lambdas and Function Type
- Move Semantics

Smart Pointers

- Standard library template pointer classes
- Aims to solve the problems associated with raw pointers management
- shared_ptr – reference counts the pointer
- unique_ptr – only allows a single copy of the pointer

Smart Pointers

- **Example:**

```
class foo {
public:
    foo ()
      : m_ptr(new bar()) {}
    ~foo () {}
private:
    bar *m_ptr;
};
```

Oops, the destructor isn't deleting the raw pointer!

Smart Pointers

- **Example:**

```
class foo {
public:
    foo ()
      : m_ptr(new bar()) {}
    ~foo () {}
private:
    std::shared_ptr<bar> m_ptr;
};
```

The shared_ptr handles deleting the pointer automatically

Agenda

- ✔ Default and Deleted Methods
- ✔ Static Assertions
- ✔ Delegated and Inherited Constructors
- ✔ Null Pointer Type
- ✔ Enum Classes
- ✔ Automatic Type Deduction
- ✔ Ranged For Loops
- ✔ Smart Pointers
- Lambdas and Function Type
- Move Semantics

Lambdas and Function Type

- Anonymous functions!
  - [] - variable capture ([&], [=], [this])
  - () - parameters
  - {} - function body
- Function type
  - 'std::function<void(int)>'

Lambdas and Function Type

- **Example:**

```
auto getSize = [](std::vector<int> vec) { return vec.size(); };

int sum = 0;
for (auto v : vec) { sum += v.get_size(); };
```

```
foo f;
auto handleFoo = [&](int i) { f.func(i); };

handleFoo(17);
```

Agenda

- ✔ Default and Deleted Methods
- ✔ Static Assertions
- ✔ Delegated and Inherited Constructors
- ✔ Null Pointer Type
- ✔ Enum Classes
- ✔ Automatic Type Deduction
- ✔ Ranged For Loops
- ✔ Smart Pointers
- ✔ Lambdas and Function Type
- Move Semantics

Move Semantics

- L-value
  - An expression with identity, that has address-able memory
  - Variables, pointers, references, parameters
- R-value
  - An expression with no identity, that does not have address-able memory
  - Literals, temporaries

Move Semantics

- **Example:**

```
int i;

int *p;

int &r;

std::string s;

void foo(int m, const int c);
```

These are all l-values

Move Semantics

- **Example:**

```
7==x

'c'

“hello world”

false

x = (y * z)
```

These are all r-values

Move Semantics

- L-value reference
  - int &r;
- R-value reference
  - int &&r;
- std::move()
  - Prolongs an r-value reference

Move Semantics

- **Example:**

```
foo make_foo() {
  foo tmp;
  tmp.init();
  return tmp;
}
```

When make_foo returns, a temporary foo object is created on the stack which is used to construct f

```
foo f = make_foo();
```

Move Semantics

- **Example:**

```
foo make_foo() {
  foo tmp;
  tmp.init();
  return std::move(tmp);
}
```

By using std::move() the move constructor for foo is triggered therefore avoiding the copy

```
foo f = make_foo();
```

Important to note that when using move semantics, the previous object becomes invalid

What Next?

- C++11 is awesome
  - Try it out
  - There are many other features
- C++14 is now out
  - Try that out too