FUNCTION OBJECTS

Function Objects by Prasanna Ghali

Plan for Today

- Different things can be used as functions in C++
- Creating generic function objects
- What lambdas are, and how they relate to ordinary function objects
- Creating prettier function objects
- What std::function is and when to use it

Function Objects

- Have always existed in C++
- Called functionals or functors
- Objects of class that defines operator ()

```
class X {
public:
  // define function call operator
  return-value operator() (arguments) const;
  ...
};
```

```
X func;
...
// a function call
func(arg1, arg2);
```

Why Function Objects?

- Functions with state
- Each function object has its own type
  - This type can be passed as template parameter
- Usually faster than function pointers
- See *wfo.cpp*

Types of Function Objects

- Zero parameter is called generator
  - See *gen.cpp*
- One parameter is called unary function
  - See *unary.cpp*
- Two parameters is called binary function
  - See *binary.cpp*
- Predicates are stateless function objects that return Boolean value
  - See *predicate.cpp*

Pass By Value

- By default, function objects are passed by value rather than by reference
- Advantage: You can pass constant and temporary expressions

```
IncreasingNumberGenerator seq(3);
std::list<int> li;
// insert sequence beginning with 3
std::generate_n(std::back_inserter(li), 5, seq);
// insert sequence beginning with 3 again ...
std::generate_n(std::back_inserter(li), 5, seq);
```

Default Pass By Value

- By default, function objects are passed by value rather than by reference
- Disadvantage: You can’t get back modifications to state of function objects
- Three ways to get result from function objects passed to algorithms:
  - Keep state externally and let function object refer to it
  - Pass function objects by reference
  - Use return value of `for_each` algorithm

Pass By Reference


```
// passing fuction objects by reference ...
IncreasingNumberGenerator seq(3);
std::list<int> li;
// insert sequence beginning with 3
std::generate_n<std::back_insert_iterator<std::list<int>>,
                         int, IncreasingNumberGenerator&>
              (std::back_inserter(li), 5, seq);
print(li, "li: ");
// insert sequence beginning with 8 again ...
std::generate_n(std::back_inserter(li), 5, seq);
```

Return Value of `for_each`

- See *foreach.cpp*

Lambdas

- So far, functions passed to algorithms already exist outside function you're using algorithms in
- Writing a proper function or whole class is tedious and possibly sign of bad software design
- Lambdas solve this problem
  - Syntactic sugar for creating unnamed function objects
  - Allow you to create function objects inline – at the place where you want them – instead of outside function you're currently writing
  - See *lambda0.cpp*

Lambda: Basic Syntax

- Syntactically, lambda expressions have 3 main parts: a head, an argument, and the body

Arguments

```
[a, &b] (int x, int y) { return a*x + b*y; }
```

Head

Body

`mutable`	(optional)
`constexpr`	(optional)
`exception attr`	(optional)
`->` *return type*	(optional)

Lambdas: Basic Syntax


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

// look for 3 ...
int three = 3;
int num_threes = std::count(v.begin(), v.end(), three);
// num_threes is 1

// look for values larger than three
auto is_above_3 = [](int v) { return v > 3; };
int num_above_3 = std::count_if(std::begin(v), std::end(v),
                                is_above_3);
std::cout << "num_above_3: " << num_above_3 << "\n";
```

Lambdas: Basic Syntax


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

// look for 3 ...
int three = 3;
int num_threes = std::count(v.begin(), v.end(), three);
// num_threes is 1

// look for values larger than three
int num_above_3 = std::count_if(std::begin(v), std::end(v),
                               [](int v) { return v > 3; });
std::cout << "num_above_3: " << num_above_3 << "\n";
```

stateless lambdas

Lambda Syntax: Head

- Specifies which variables from surrounding scope will be visible inside lambda body
- Variables can be captured as values or by references

Arguments

```
[a, &b] (int x, int y) { return a*x + b*y; }
```

Head

Body

Lambda Syntax: Head

- `[a, &b]` – a is captured by value; b by reference
- `[]` – nothing from outer scope is used
- `[&]` – outer scope variables are passed by reference
- `[=]` – outer scope variables are passed by value
- `[this]` – capture `this` pointer by value
- `[&, a]` – outer scope variables are passed by value, except a, which is captured by value
- `[=, &b]` – outer scope variables are passed by value, except b, which is passed by reference

Lambdas: Capture Clause


```
int count_value_above(std::vector<int> const& v, int x) {
  auto is_above = [x](int i) { return i > x; };
  return std::count_if(std::begin(v), std::end(v),
                       is_above);
}
```

```
int count_value_above(std::vector<int> const& v, int x) {
  auto is_above = [&x](int i) { return i > x; };
  return std::count_if(v.begin(), v.end(), is_above);
}
```

Capture by Value Versus Capture by Reference


```
std::vector<int> vi{1,2,3,4,5,6};
int x = 3;
auto is_above = [x](int v) {
  return v > x;
};
x = 4;
int count_b = std::count_if(
    std::begin(vi),
    std::end(vi),
    is_above
    );  // count_b is what value?
```

```
std::vector<int> vi{1,2,3,4,5,6};
int x = 3;
auto is_above = [&x](int v) {
  return v > x;
};
x = 4;
int count_b = std::count_if(
    std::begin(vi),
    std::end(vi),
    is_above
    );  // count_b is what value?
```

Lambdas: Under the Hood [Capture by Value]


```cpp
int x {3};

auto is_above = [x](int y) {
  return y > x;
};

bool test = is_above(5);
```

```cpp
int x {3};

class IsAbove {
public:
  IsAbove(int vx) : x{vx} {}
  auto operator()(int y) const {
    return y > x;
  }
private:
  int x{}; // Value
};

IsAbove is_above{x};
bool test = is_above(5);
```

Lambdas: Under the Hood [Capture by Reference]


```cpp
int x {3};

auto is_above = [&x](int y) {
  return y > x;
};

bool test = is_above(5);
```

```cpp
int x {3};

class IsAbove {
public:
  IsAbove(int& rx) : x{rx} {}
  auto operator()(int y) const {
    return y > x;
  }
private:
  int &x; // Value
};

IsAbove is_above{x};
bool test = is_above(5);
```

Initializing Variables in Capture


```
auto some_func =
  [numbers = std::list<int>{4,2}]() {
  for (int i : numbers) {
    std::cout << i;
  }
};

some_func();  // output: 42
```

Initializing Variables in Capture


```cpp
auto some_func =
  [numbers = std::list<int>{4,2}]() {
  for (int i : numbers) {
    std::cout << i;
  }
};

some_func(); // output: 42
```

```cpp
class SomeFunc {
public:
  SomeFunc() : numbers{4, 2} {}
  void operator()() const {
    for (int i : numbers) {
      std::cout << i;
    }
  }
private:
  std::list<int> numbers;
};

SomeFunc some_func{};
some_func(); // Output: 42
```

Initializing Variables in Capture


```
int x {1};
auto some_func = [&y = x]() {
  // y is a reference to x
};
```

```
std::unique_ptr<int> x {std::make_unique<int>()};
auto some_func = [y = std::move(x)]() {
  // Use x here..
};
```

Mutating Lambda Variables


```
auto counter =  [count=10] () mutable {
  return ++count;
};

for (size_t i{}; i < 5; ++i) {
  std::cout << counter() << " ";
}
std::cout << "\n";
```

Mutating Lambda Variables


```cpp
int v {7};
auto lambda = [v]() mutable {
  std::cout << v << " ";
  ++v;
};
assert(v == 7);
lambda(); lambda();
assert(v == 7);
std::cout << v;
```

```cpp
class Lambda {
 public:
 Lambda(int m) : v{m} {}
 void operator()() {
   std::cout<< v << " ";
   ++v;
 }
private:
  int v{};
};
```

Mutating Lambda Variables


```cpp
int v {7};
auto lambda = [&v]() {
  std::cout << v << " ";
  ++v;
};
assert(v == 7);
lambda();
lambda();
assert(v == 9);
std::cout << v;
```

```cpp
class Lambda {
public:
  Lambda(int& m) : v{m} {}
  auto operator()() const {
    std::cout<< v << " "; ++v;
  }
private:
  int& v;
};
```

Capture All


```
class Foo {
public:
  void member_function() {
    int a {0};
    float b {1.0f};
    // capture all variables by copy
    auto lambda0 = [=]() {std::cout << a << b;};
    // capture all variables by reference
    auto lambda1 = [&]() {std::cout << a << b;};
    // capture entire object by reference
    auto lambda2 = [this]() {std::cout << m ;};
    // capture object by copy
    auto lambda3 = [*this]() {std::cout << m;};
 }
private:
  int m {};
};
```

Capture All

- Using [=] or [&] doesn't mean all variables in scope are copied into lambda
  - Only variables actually used inside lambda are copied
- Although convenient to capture all variables with [=] or [&], never a good idea
  - Performance penalties
  - Easier to interpret when specific identifiers labeled in capture list

Capture All

- When capturing all variables by value [reference], you can specify variables to be captured by reference [value]

```
// capture a, b, c by value
auto l1 = [=] { /* ... */ ; };
// capture a, b, c by reference
auto l2 = [&] { /* ... */ ; };
// capture a and b by value and c by reference
auto l3 = [=, &c] { /* ... */ ; };
// capture a and b by reference and c by value
auto l4 = [&, =c] { /* ... */ ; };
```

Capture All

- Although convenient to capture all variables with [=] or [&], never a good idea
  - Performance penalties
  - Easier to interpret when specific identifiers labeled in capture list

Capture-Less Lambdas and Function Pointers

- Lambdas without capture can be implicitly converted to function pointers

```
extern void press_button(char const *msg,
     void (*callback)(int, char const*));

// + indicates lambda has no captures
auto lambda = +[](int result, char const *str) {
  // process result and str
};

press_button("pressed", lambda);
```

What About Lambdas With Capture?

- Lambdas with captures have their own unique type
- Even if two lambdas with capture are plain clones of each other, they still have their own unique type

Need for std::function<>

- Function template for_upto can be used used with any callable (lambda, function pointer, function object)

```
template <typename F>
void for_upto(int N, F f) {
  for (int i{0}; i <= N; ++i) {
    f(i);
  }
}
```

Need for std::function<>

- Relies on automatic type deduction to take lambda or function pointer …

```
template <typename T>
void print(T t) { std::cout << t << ' '; }

// insert values from 0 to 4
std::vector<int> values;
for_upto(5, [&values](int i) { values.push_back(i); });

// print elements
for_upto(5, print<int>); printf("\n");
```

Need for `std::function<>`

- Each use of `for_upto` will likely produce different instantiation
  - If `for_upto` was large, possible for its instantiations to increase code size
- One approach to limit increase in code size is to turn function template into non-template

```
void for_upto(int N, void (*f)(int)) {
  for (int i{0}; i != N; ++i) {
    f(i);
  }
}
```

Need for `std::function<>`

- However, it'll produce an error when passed a capture lambda

```
void for_upto(int N, void (*f)(int)) {
  for (int i{0}; i != N; ++i) {
    f(i);
  }
}

// this lambda implicitly converts to function pointer
for_upto(5, +[](int i) { std::cout << i << " "; });

// this lambda will not convert to function pointer
for_upto(5, [&values](int i) { values.push_back(i); });
```

std::function<>

- Standard library's class template std::function<> permits alternative formulation of for_upto

Template parameter of std::function<> specifies return type of function and its arguments.
Can store anything having signature specified in std::function<> template parameter

```
void for_upto(int N, std::function<void(int)> f) {
  for (int i{0}; i != N; ++i) { f(i); }
}
```

std::function<>


```
std::vector<std::function<float(float,float)>> tasks;
tasks.push_back(std::fmaxf); // plain function
tasks.push_back(std::multiplies<float>()); // function object
tasks.push_back(std::multiplies<>()); // generic call operator

float x = 1.1f;
tasks.push_back([x](float a, float b) { return a*x+b; }); // lambda
tasks.push_back(
     [x](auto a, auto b) { return a*x+b; }); // generic lambda

// call each task
for (std::function<float(float, float)> f : tasks) {
  std::cout << f(10.1, 11.1) << "\n";
}
```

std::function<>: Can Do More

- Normally, C++ core language stops you from calling member function as non-member function (`std::string::length(str)`)
- Can do if member function stored in `function<>` object

```
std::string str{"C++: terror or horror"};
std::function<std::string::size_type(std::string&)> f;
f = &std::string::length;
std::cout << f(str);
```

std::function<>: Can Do More


```
class C {
public:
  int func(int x, int y) const { return x*y; }
};

std::function<int(C const&, int, int)> f;
f = &C::func;
std::cout << mf(C(), 10, 20) << "\n";
```

Properties

- Generalized form of C++ function pointer with some fundamental operations
  - Can be used to invoke function without caller knowing anything about function itself
  - Can be copied, moved, and assigned
  - Can be initialized or assigned from another function (with compatible signature)
  - Has null state that indicates when no function is bound to it

Performance Considerations

- Generalized form of C++ function pointer with some fundamental operations
  - Lambdas can be inlined but not functions that're wrapped in `std::function<>`
  - `std::function<>` may use heap-allocated memory to store captured variables
  - Additional run-time overhead involved with calling functions wrapped in `std::function<>`

Generic Lambdas

- Generic lambdas can deduce their types not captured values
- Function call operator becomes member function template

```
auto v {3}; // int
auto lambda = [v] (auto v0, auto v1) {
  return v + v0*v1;
};
```

Generic Lambdas: Under the Hood


```
auto v {3}; // int
auto lambda =
 [v] (auto v0, auto v1) {
  return v + v0*v1;
};
```

```
class Lambda {
public:
  Lambda(int x) : v{x} {}
  template <typename T0, typename T1>
  auto operator()(T0 v0, T1 v1) const
  {
    return v + v0*v1;
  }
private:
  int v{};
};

auto v = 3;
auto lambda = Lambda{v};
```

Generic Lambdas

- Just like templated version, compiler won't generate actual function until lambda is invoked

```
// calls
auto lambda_int = lambda(1, 2);
auto lambda_dbl = lambda(1.0, 2.0);
```

```
auto lambda_int = [v](int v0, int v1) { return v + v0*v1; };
auto lambda_dbl = [v](double v0, double v1) { return v + v0*v1; };


auto res_int = lambda_int(1, 2);
auto res_dbl = lambda_dbl(1.0, 2.0);
```

Partial Function Application

- Create new callable from existing one by fixing one or more of its arguments to specific value
- Partial means you provide some, but not all, arguments needed to calculate result of function
- C++11 provides `std::bind` to implement partial function application

Things To Do With `std::bind`

- Bind arguments to arbitrary position
- Change order of arguments
- Introduce placeholders for arguments
- Partially evaluate algorithms
- Invoke newly created function objects, use them in algorithm, or store them in `std::function`

Using Lambdas As Alternative To `std::bind`

- `std::bind` comes with cost of making code harder to optimize
- Turning all `std::bind` calls to lambda is simple
  - Turn any argument bound to variable or reference to variable into captured variable
  - Turn all placeholders into lambda arguments
  - Specify arguments bound to specific value directly in lambda body