FUNCTION OBJECTS

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`

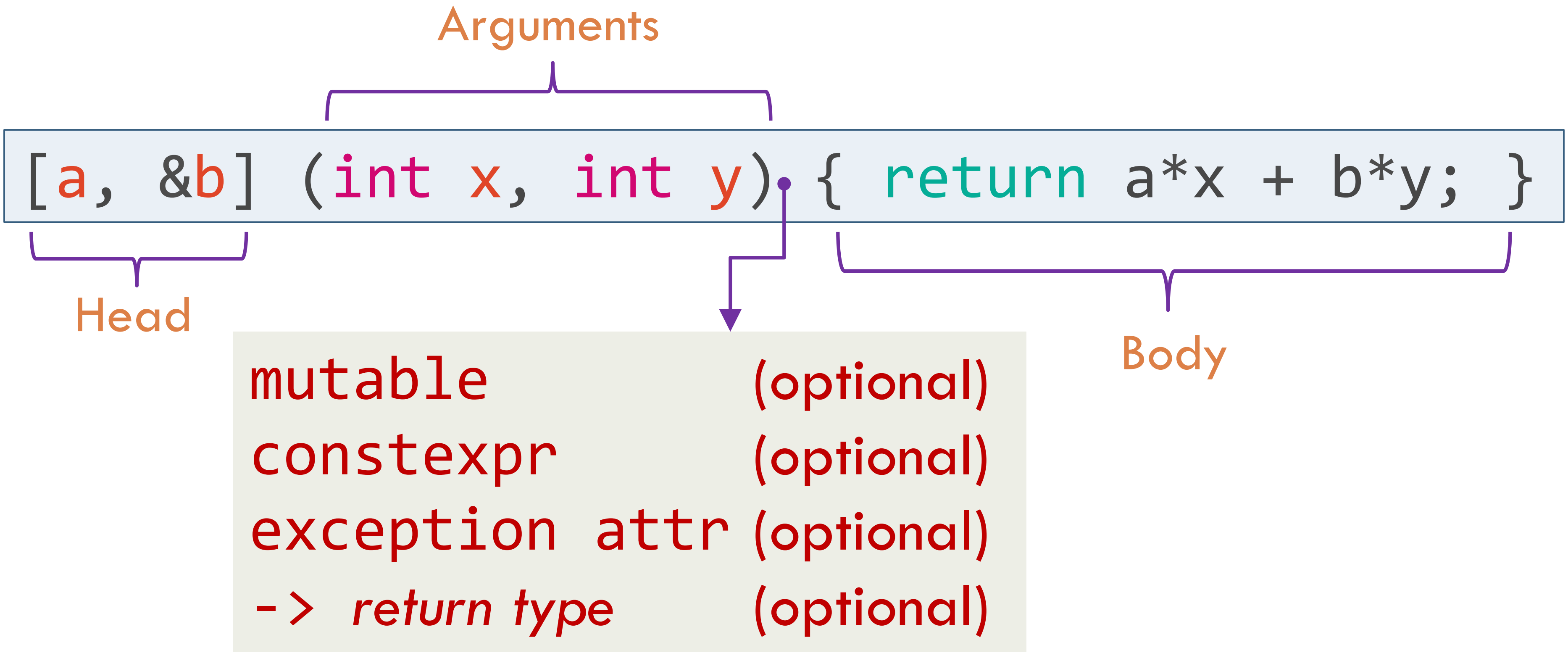
- See *foreach.cpp*

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

mutable (optional)

constexpr (optional)

exception attr (optional)

-> *return type* (optional)

Body

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";
```

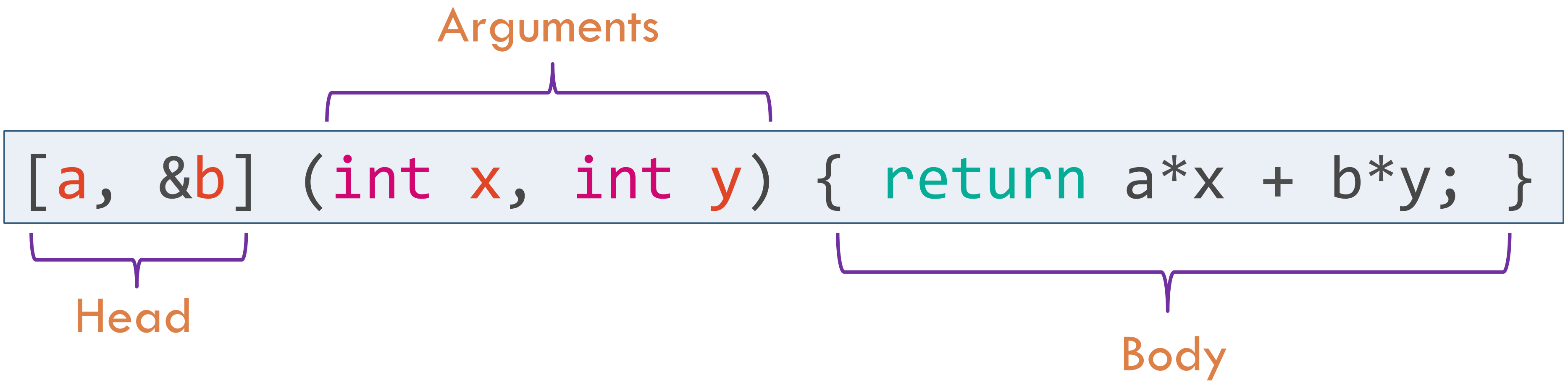
Lambdas: Basic Syntax

```
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



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

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

```cpp
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?
```

```cpp
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]

```
int x {3};

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

bool test = is_above(5);
```

```
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

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

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

```
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

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

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

Mutating Lambda Variables

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

```
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 {};
};
```

Lambdas and Function Pointers

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

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