LAMBDAS IN C++

LECTURE 11-2

JIM FIX, REED COLLEGE CS2-S20

REMINDER: 2ND MIDTERM THURSDAY

- **Midterm** Thursday-Friday on circuits and MIPS assembly
  - Will post PDF in Git repo on as an assignment link Thursday by noon PST
  - Designed as an in-class 80 minute exam with 6 problems
  - Submit images and/or text files within repo on GitHub
    - ➡ Use about 2 hours total to take the exam and to assemble/submit work
  - Must be submitted Friday by noon PST
  - Treat as a written exam. E.g. don't need to compile/test.

RECALL: USED A FUNCTION OBJECT IN SORT

```
// Build a list of word/frequency pairs from a dictionary.
std::vector<std::pair<std::string,int>> ws { };
for (auto p = d.begin(); p != d.end(); p++) {
  ws.push_back(*p);
}

// Sort in order of decreasing frequency.
auto compare = [](std::pair<std::string,int> entry1,
                  std::pair<std::string,int> entry2) -> bool
               { return entry1.second > entry2.second; };
//
std::sort(ws.begin(), ws.end(), compare);

// Output the top 100.
std::cout << "The 100 most-used words are:" << std::endl;
for (int i=0; i < 100; i++) {
  std::cout << ws[i].first << ":" << ws[i].second << std::endl;
}
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

`[` *captures* `](` *parameters* `) -> ` *result* ` { ` *rule* ` }`

- *captures*: list of variables from the context that are used *by value* or *by reference* in the rule.
- *parameters*: the function's parameters and their types
- *result*: type of the returned result
- *rule*: code that computes the return result from the parameters; the "body"

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones} -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

[*captures*] (*parameters*) -> *result* { *rule* }

- *captures*: list of variables from the context that are used *by value* or *by reference* in the rule. We'll look at this soon...
- *parameters*: the function's parameters and their types
- *result*: type of the returned result
- *rule*: code that computes the return result from the parameters; the "body"

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones} -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

[*captures*] (*parameters*) -> *result* { *rule* }

- *captures*: list of variables from the context that are used *by value* or *by reference* in the rule.
- *parameters*: the function's parameters and their types
- *result*: type of the returned result
- *rule*: code that computes the return result from the parameters; the "body"

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones} -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

`[` *captures* `](` *parameters* `) -> result { ` *rule* ` }`

- *captures*: list of variables from the context that are used *by value* or *by reference* in the rule.
- *parameters*: the function's parameters and their types
- *result*: type of the returned result
- *rule*: code that computes the return result from the parameters; the "body"

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones} -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

[*captures*] (*parameters*) -> *result* { *rule* }

- *captures*: list of variables from the context that are used *by value* or *by reference* in the rule.
- *parameters*: the function's parameters and their types
- *result*: type of the returned result
- *rule*: code that computes the return result from the parameters; the "body"

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones} -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

[*captures*] (*parameters*) -> *result* { *rule* }

- *captures*: list of variables from the context that are used *by value* or *by reference* in the rule.
- *parameters*: the function's parameters and their types
- *result*: type of the returned result
- *rule*: code that computes the return result from the parameters; the "body"

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones} -> int { return 10*tens+ones; } // two_digit
```

NOTE: can span multiple lines.

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                  { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                  { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                    std::cout << n << std::endl;
                                    std::cout << n << std::endl;
                                    std::cout << n << std::endl;
                                  };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                  { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                  { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                    std::cout << n << std::endl;
                                    std::cout << n << std::endl;
                                    std::cout << n << std::endl;
                                  };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                  { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                  { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                    std::cout << n << std::endl;
                                    std::cout << n << std::endl;
                                    std::cout << n << std::endl;
                                  };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -
                                     std::cout << n << s
                                     std::cout << n << s
                                     std::cout << n << s
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

BTW, this outputs:

```
1
11
5
54321
54321
54321
```

EXAMPLE: COMPARISON FUNCTION OBJECT FOR SORT

Here is the syntax for an anonymous function object:

`[` *captures* `](` *parameters* `) -> ` *result* ` { ` *rule* ` }`

- *parameters*: the function's parameters and their types
- *result*: type of the returned result
- *rule*: code that computes the return result from the parameters; the "body"

Examples:

```
[](std::pair<std::string,int> entry1,
   std::pair<std::string,int> entry2) -> bool
                { return entry1.second > entry2.second; }
```

EXAMPLE: COMPARISON FUNCTION OBJECT FOR SORT

Here is the syntax for an anonymous function object:

[*captures*] (*parameters*) -> *result* { *rule* }

- *parameters*: the function's parameters and their types
- *result*: type of the returned result
- *rule*: code that computes the return result from the parameters; the "body"

Examples:

```
[](std::pair<std::string,int> entry1,
   std::pair<std::string,int> entry2) -> bool
                { return entry1.second > entry2.second; }
```

EXAMPLE: COMPARISON FUNCTION OBJECT FOR SORT

Here is the syntax for an anonymous function object:

[*captures*] (*parameters*) -> *result* { *rule* }

- *parameters*: the function's parameters and their types
- *result*: type of the returned result
- *rule*: code that computes the return result from the parameters; the "body"

Examples:

```
[](std::pair<std::string,int> entry1,
   std::pair<std::string,int> entry2) -> bool
                { return entry1.second > entry2.second; }
```

EXAMPLE: COMPARISON FUNCTION OBJECT FOR SORT

Here is the syntax for an anonymous function object:

[*captures*] (*parameters*) -> *result* { *rule* }

- *parameters*: the function's parameters and their types
- *result*: type of the returned result
- *rule*: code that computes the return result from the parameters; the "body"

Examples:

```
[](std::pair<std::string,int> entry1,
   std::pair<std::string,int> entry2) -> bool
                { return entry1.second > entry2.second; }
```

NOTICE THE TYPES

```
std::function<bool(int)> isEven = [](int i) -> bool
                          { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                          { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                          { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                            std::cout << n << std::endl;
                            std::cout << n << std::endl;
                            std::cout << n << std::endl;
                          };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

NOTICE THE TYPES

At the top of this code:

```
#include <functional>
```

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

SYNTAX FOR A FUNCTION OBJECT'S TYPE

Here is the syntax for an anonymous function object's type:

`std::function<` *result-type* `(` *types-of-parameters* `)>`

- *types-of-parameters*: the function's parameter types
- *result-type*: type of the returned result
- Example:

```
std::function<void(bool,std::string)> maybePrint =
  [](bool yes, std::string message) -> void {
    if (yes) {
      std::cout << message << std::endl;
    };
std::function<int(int,int)> two_digit =
  [](int tens_digit, int ones_digit) -> {
    return tens_digit*10 + ones_digit;
  }
```

SYNTAX FOR A FUNCTION OBJECT'S TYPE

Here is the syntax for an anonymous function object's type:

`std::function<` *result-type* `(` *types-of-parameters* `)>`

- *types-of-parameters*: the function's parameter types
- *result-type*: type of the returned result
- Example:

```
std::function<void(bool,std::string)> maybePrint =
  [](bool yes, std::string message) -> void {
    if (yes) {
      std::cout << message << std::endl;
    };
std::function<int(int,int)> two_digit =
  [](int tens_digit, int ones_digit) -> {
    return tens_digit*10 + ones_digit;
  }
```

CAPTURE BY VALUE

- Unlike named C++ functions, anonymous functions are defined within the context of executable code.
  - stack variables are available, stack objects are available
  - pointers to heap objects are available
- Can indicate that these things can be accessed by the function object.

Example:

```
int tens_digit = 5;
std::function<int(int)> make_fifty_something =
  [tens_digit](int ones_digit) -> int {
     return tens_digit * 10 + ones_digit;
  };
std::cout << make_fifty_something(8) << std::endl;
std::cout << make_fifty_something(7) << std::endl;
```

CAPTURE BY VALUE

- Unlike named C++ functions, anonymous functions are defined within the context of executable code.
  - stack variables are available, stack objects are available
  - pointers to heap objects are available
- Can *indicate that these things can be accessed* by the function object.

Example:

```
int tens_digit = 5;
std::function<int(int)> make_fifty_something =
  [tens_digit](int ones_digit) -> int {
     return tens_digit * 10 + ones_digit;
  };
std::cout << make_fifty_something(8) << std::endl;
std::cout << make_fifty_something(7) << std::endl;
```

CAPTURE BY VALUE

- Unlike named C++ functions, anonymous functions are defined within the context of executable code.
  - stack variables are available, stack objects are available
  - pointers to heap objects are available
- Can indicate that these things can be accessed by the function object.

Example:

```
int tens_digit = 5;
std::function<int(int)> make_fifty_something =
  [tens_digit](int ones_digit) -> int {
     return tens_digit * 10 + ones_digit;
  };
std::cout << make_fifty_something(8) << std::endl;
std::cout << make_fifty_something(7) << std::endl;
```

This outputs:

```
58
57
```

CAPTURE BY VALUE (CONT'D)

- NOTE: the variable is copied *by value*.
  - ➡ Subsequent changes aren't reflected because of this kind of capture.

Example:

```
int tens_digit = 5;
std::function<int(int)> make_fifty_something =
  [tens_digit](int ones_digit) -> int {
     return tens_digit * 10 + ones_digit;
  };
tens_digit--;
std::cout << make_fifty_something(8) << std::endl;
tens_digit--;
std::cout << make_fifty_something(7) << std::endl;
```

CAPTURE BY VALUE (CONT'D)

- NOTE: the variable is copied *by value*.
  - ➡ Subsequent changes aren't reflected because of this kind of capture.

Example:

```
int tens_digit = 5;
std::function<int(int)> make_fifty_something =
  [tens_digit](int ones_digit) -> int {
     return tens_digit * 10 + ones_digit;
  };
tens_digit--;
std::cout << make_fifty_something(8) << std::endl;
tens_digit--;
std::cout << make_fifty_something(7) << std::endl;
```

CAPTURE BY VALUE (CONT'D)

- NOTE: the variable is copied *by value*.
  - ➡ Subsequent changes aren't reflected because of this kind of capture.

Example:

```
int tens_digit = 5;
std::function<int(int)> make_fifty_something =
  [tens_digit](int ones_digit) -> int {
     return tens_digit * 10 + ones_digit;
  };
tens_digit--;
std::cout << make_fifty_something(8) << std::endl;
tens_digit--;
std::cout << make_fifty_something(7) << std::endl;
```

Also outputs:

```
58
57
```

FUN WITH VALUE CAPTURE

- I can invent "higher-order functions" that return functions as values.

Example:

```
std::function<int(int)> makeAdder(int dx) {
  return [dx](int x) { return x+dx; };
}

std::function<void()> makePrinter(int x) {
  return [x]() { std::cout << x << "\n"; };
}

int main(void) {
  std::function<int(int)> add10 = makeAdder(10);
  std::cout << "add10(5) = " << add10(5) << std::endl;

  std::function<void()> print5 = makePrinter(5);
  std::cout << "print5(): " << std::endl;
  print5();
}
```

FUN WITH VALUE CAPTURE

- I can invent "higher-order functions" that return functions as values.

Example:

```
std::function<int(int)> makeAdder(int dx) {
  return [dx](int x) { return x+dx; };
}

std::function<void()> makePrinter(int x) {
  return [x]() { std::cout << x << "\n"; };
}

int main(void) {
  std::function<int(int)> add10 = makeAdder(10);
  std::cout << "add10(5) = " << add10(5) << std::endl;

  std::function<void()> print5 = makePrinter(5);
  std::cout << "print5(): " << std::endl;
  print5();
}
```

FUN WITH VALUE CAPTURE

- I can invent "higher-order functions" that return functions as values.

Example:

```
std::function<int(int)> makeAdder(int dx) {
  return [dx](int x) { return x+dx; };
}

std::function<void()> makePrinter(int x) {
  return [x]() { std::cout << x << "\n"; };
}

int main(void) {
  std::function<int(int)> add10 = makeAdder(10);
  std::cout << "add10(5) = " << add10(5) << std::endl;

  std::function<void()> print5 = makePrinter(5);
  std::cout << "print5(): " << std::endl;
  print5();
}
```

CAPTURE BY REFERENCE

- Alternatively, you can *indicate that variables' values can be tracked and altered* by the function object.
- You indicate this with the by-reference annotation **&**.

Example:

```
int tens_varies = 5;
std::function<int(int)> make_umpty_something =
  [&tens_varies](int ones_digit) -> int {
     return tens_varies * 10 + ones_digit;
  };
tens_varies--;
std::cout << make_umpty_something(8) << std::endl;
tens_varies--;
std::cout << make_umpty_something(7) << std::endl;
```

CAPTURE BY REFERENCE

- Alternatively, you can *indicate that variables' values can be tracked and altered* by the function object.
- You indicate this with the by-reference annotation **&**.

Example:

```
int tens_varies = 5;
std::function<int(int)> make_umpty_something =
  [&tens_varies](int ones_digit) -> int {
     return tens_varies * 10 + ones_digit;
  };
tens_varies--;
std::cout << make_umpty_something(8) << std::endl;
tens_varies--;
std::cout << make_umpty_something(7) << std::endl;
```

Outputs:

```
48
37
```

HERE ARE SOME USES OF REFERENCE

```
int count = 0;
std::function<void(void)> increment =
      [&count](void) -> void { count++; }
int x = 10;
int y = 11;
std::function<void(void)> increment =
      [&x,&y](void) -> void {
  int tmp = x;
  x = y;
  y = tmp;
};
increment();
std::cout << count << " " << x << " " << y << std::endl;
increment();
swap();
std::cout << count << " " << x << " " << y << std::endl;
increment();
swap();
std::cout << count << " " << x << " " << y << std::endl;
```

HERE ARE SOME USES OF REFERENCE

```
int count = 0;
std::function<void(void)> increment =
      [&count](void) -> void { count++; }
int x = 10;
int y = 11;
std::function<void(void)> increment =
      [&x,&y](void) -> void {
  int tmp = x;
  x = y;
  y = tmp;
};
increment();
std::cout << count << " " << x << " " << y << std::endl;
increment();
swap();
std::cout << count << " " << x << " " << y << std::endl;
increment();
swap();
std::cout << count << " " << x << " " << y << std::endl;
```

HERE ARE SOME USES OF REFERENCE

```
int count = 0;
std::function<void(void)> increment =
      [&count](void) -> void { count++; }
int x = 10;
int y = 11;
std::function<void(void)> increment =
      [&x,&y](void) -> void {
  int tmp = x;
  x = y;
  y = tmp;
};
increment();
std::cout << count << " " << x << " " << y << std::endl;
increment();
swap();
std::cout << count << " " << x << " " << y << std::endl;
increment();
swap();
std::cout << count << " " << x << " " << y << std::endl;
```

This outputs:

```
1 10 11
2 11 10
3 10 11
```

HERE ARE SOME USES OF REFERENCE (CONT'D)

```
std::vector<int> v = {12, 8, 17};
std::function<void(void)> output =
  [&v](void) -> void {
    for (int e : v) {
      std::cout << e << " ";
    }
    std::cout << std::endl;
  };
std::function<void(int,int)> change =
   [&v](int index, int value) -> void { v.at(index) = value; };

output();
change(2,37);
output();
change(0,32);
output();
change(1,3);
output();
```

HERE ARE SOME USES OF REFERENCE (CONT'D)

```
std::vector<int> v = {12, 8, 17};
std::function<void(void)> output =
  [&v](void) -> void {
    for (int e : v) {
      std::cout << e << " ";
    }
    std::cout << std::endl;
  };
std::function<void(int,int)> change =
   [&v](int index, int value) -> void { v.at(index) = value; };

output();
change(2,37);
output();
change(0,32);
output();
change(1,3);
output();
```

HERE ARE SOME USES OF REFERENCE (CONT'D)

```
std::vector<int> v = {12, 8, 17};
std::function<void(void)> output =
  [&v](void) -> void {
    for (int e : v) {
      std::cout << e << " ";
    }
    std::cout << std::endl;
  };
std::function<void(int,int)> change =
   [&v](int index, int value) -> void { v.at(index) = value; };

output();
change(2,37);
output();
change(0,32);
output();
change(1,3);
output();
```

HERE ARE SOME USES OF REFERENCE (CONT'D)

```
std::vector<int> v = {12, 8, 17};
std::function<void(void)> output =
  [&v](void) -> void {
    for (int e : v) {
      std::cout << e << " ";
    }
    std::cout << std::endl;
  };
std::function<void(int,int)> change =
   [&v](int index, int value) -> void { v.at(index) = value; };

output();
change(2,37);
output();
change(0,32);
output();
change(1,3);
output();
```

This outputs:

```
12 8 17
12 8 37
32 8 37
32 3 37
```

USE WITHIN ALGORITHMS PACKAGE

```
std::vector<int> v {0,1,4,9,16,25,36,49,64};
int sum = 0;
std::for_each(v.begin(),v.end(),
              [&sum](int e) mutable -> void { sum +=e; } );
std::cout << "sum = " << sum << std::endl;
// outputs sum = 444
```

USE WITHIN ALGORITHMS PACKAGE

```
std::vector<int> v {0,1,4,9,16,25,36,49,64};
int sum = 0;
std::for_each(v.begin(),v.end(),
              [&sum](int e) mutable -> void { sum +=e; } );
std::cout << "sum = " << sum << std::endl;
// outputs sum = 444
```

HMM... WILL TALK ABOUT THIS LATER

CAPTURE LIST

RECALL the syntax for an anonymous function object:

`[` *capture-list* `]` `(` *parameters* `)` `->` *result* `{` *rule* `}`

- *capture-list*: list of variables from the context that are used in the rule
- Can be used *by value* or *by reference*:
  - If you want the variable's value to be copied use no annotation
  - If you want the stack object/variable to be referenced, changed, use **&**

WHAT C++ BUILDS

The function object returned by **`makeAdder`**, shown just below...

```
std::function<int(int)> makeAdder(int dx) {
  return [dx](int x) { return x+dx; };
}
```

...is essentially an instance of this class

```
class Foo {
private:
  int toAdd;
public:
  Foo(int dx) : toAdd {dx} {}
  int operator()(int x) const { return x+toAdd; }
};
```

WHAT C++ BUILDS

The function object returned by **`makeAdder`**, shown just below...

```
std::function<int(int)> makeAdder(int dx) {
  return [dx](int x) { return x+dx; };
}
```

...is essentially an instance of this class

```
class Foo {
private:
  int toAdd;
public:
  Foo(int dx) : toAdd {dx} {}
  int operator()(int x) const { return x+toAdd; }
};
```

WHAT C++ BUILDS

The function object returned by **`makeAdder`**, shown just below...

```
std::function<int(int)> makeAdder(int dx) {
  return [dx](int x) { return x+dx; };
}
```

...is essentially an instance of this class

```
class Foo {
private:
  int toAdd;
public:
  Foo(int dx) : toAdd {dx} {}
  int operator()(int x) const { return x+toAdd; }
};
```

WHAT C++ BUILDS

The function object returned by **`makeAdder`**, shown just below...

```
std::function<int(int)> makeAdder(int dx) {
  return [dx](int x) { return x+dx; };
}
```

...is essentially an instance of this class

```
class Foo {
private:
  int toAdd;
public:
  Foo(int dx) : toAdd {dx} {}
  int operator()(int x) const { return x+toAdd; }
};
```

WHAT C++ BUILDS

With these definitions:

```
std::function<int(int)> makeAdder(int dx) {
  return [dx](int x) { return x+dx; };
}

class Foo {
private:
  int toAdd;
public:
  Foo(int dx) : toAdd {dx} {}
  int operator()(int x) const { return x+toAdd; }
};
```

We can write this:

```
std::cout << "(Foo {10})(5) = " << (Foo {10})(5) << std::endl;
std::cout << "(makeAdder(10))(5) = " << (makeAdder(10))(5)
          << std::endl;
```

MUTABLE KEYWORD?

When I first taught myself C++'s lambda, used *mutable* keyword

`[` *capture-list* `]` `(` *parameters* `)` **`mutable`** `->` *result* `{` *body* `}`

- This keyword is used to indicate that "the function's body can modify the variables/objects it captures."
- (It makes the **`operator()`** of the anonymous class **`const`** otherwise.)
- So, technically, my **`increment`**, **`swap`**, **`change`** examples should have been marked **`mutable`**.
- But they worked anyway in my **`samples/lambdas1.cc`**?!?

SUMMARY OF C++ LAMBDA

- C++ allows you to concisely express "function objects".
  - ➡ They are essentially one-time class instances with **`operator()`** defined.
- They are called *lambdas*
  - ➡ From the programming language Lisp: **`(lambda (n) (+ n 1))`**
  - ➡ Lisp's John McCarthy took them from Alonzo Church's "lambda calculus"
- Because C++ has a complex object memory model, must specify *captures*
  - ➡ overloads **`&`** syntax and has key word **`mutable`** to specify behavior
- Useful for many components defined in the **`algorithm`** STL