Odds and Ends

CMSC 240

All examples borrowed/modified from

C++ Crash Course by Josh Lospinoso

No Starch Press

Before a Working Example…

- Some C++ concepts that we'll need for this example
  - Function objects
  - Lambda expressions

Function Objects

- One can make user-defined types callable or invocable
  - Done by overloading the function-call operator `operator()()`
- Such a type is called a *function type*
  - Instances of a function type are *function objects*
- The function-call operator permits any combination of argument types, return types, and modifiers (except `static`)

Function Objects

- Why would you want to do this?
  - Might need to interoperate with code that expects function objects
    - Many libraries, including `stdlib` use the function call operator as interface to function-like objects (we'll see one later)
    - Ex. Creating asynchronous task with `std:asynch` function, which accepts arbitrary function object that can execute on a separate thread

Function Objects

- Why would you want to do this?
  - The designers of `std::asynch` could have required coder to expose a run method
  - But function call operator allows generic code to use identical notation to invoke a function or a function-object

```cpp
#include <cstdint>
#include <cstdio>

struct CountIf {
  CountIf(char x)
      : x{ x } {}
  size_t operator()(const char* str) const {
    size_t index{}, result{};
    while(str[index]) {
      if(str[index] == x)
        result++;
      index++;
    }
    return result;
  }

  private:
  const char x;
};

int main() {
  CountIf s_counter{ 's' };
  auto sally = s_counter("Sally sells seashells by the seashore.");
  printf("Sally: %zd\n", sally);
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zd\n", sailor);
  auto buffalo = CountIf{ 'f' }("Buffalo buffalo Buffalo buffalo "
                                "buffalo buffalo Buffalo buffalo.");
  printf("Buffalo: %zd\n", buffalo);
}
```

```cpp
#include <cstdint>
#include <cstdio>

struct CountIf {
  CountIf(char x)
      : x{ x } {}
  size_t operator()(const char* str) const {
    size_t index{}, result{};
    while(str[index]) {
      if(str[index] == x)
        result++;
      index++;
    }
    return result;
  }

  private:
  const char x;
};

int main() {
  CountIf s_counter{ 's' };
  auto sally = s_counter("Sally sells seashells by the seashore.");
  printf("Sally: %zd\n", sally);
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zd\n", sailor);
  auto buffalo = CountIf{ 'f' }("Buffalo buffalo Buffalo buffalo "
                                "buffalo buffalo Buffalo buffalo.");
  printf("Buffalo: %zd\n", buffalo);
}
```

Output:

```
Sally: 7
Sailor: 3
Buffalo: 16
```

Lambda Expressions

- *Lambda expressions* construct unnamed function objects succinctly
  - The function object implies the function type
    - Quick way to create a function object
- Can’t do anything a plain old function declaration can’t do
  - But in specific contexts can be very convenient
    - Declaring function objects can be verbose. Lambda expressions much more succinct

Lambda Expressions: Usage

- Five components
  - *captures:* member variables of the function object
  - *parameters:* arguments required to invoke function object
  - *body:* function object's code
  - *specifiers:* E.g., constexpr, noexcept
  - *return type:* just what you think

Lambda Expressions: Usage

- Syntax:
- [captures] (parameters) modifiers -> return type { body }
- Only capture and body required
  - So everything else is optional
- Each lambda component has direct analogue to part of function object…

Lambda Expressions: Usage

```
struct CountIf {
  CountIf(char x)
      : x{ x } {}
  size_t operator()(const char* str) const {
    --snip--
  }

  private:
  const char x;
};
```

Lambda Expressions: Usage

```
struct CountIf {
  CountIf(char x)
      : x{ x } {}
  size_t operator()(const char* str) const {
    --snip--
  }

  private:
  const char x;
};
```

capture

parameters

return type

body

specifiers

Lambda Parameters and Bodies

- Lambda expressions produce function objects, and thus are callable
  - You'll often want the function object to accept parameters upon invocation
- Lamba expression body is just like a function body – all parameters have function scope
- Declare lambda parameters and bodies using essentially same syntax as for functions

Lambda Parameters and Bodies

- Example:

```
[](int x){return x*x;}
```

- This lambda takes a single `int x` and uses it in the body to perform squaring

Lambda Example

```
#include <cstdint>
#include <cstdio>

template <typename Fn>
void transform(Fn fn, const int* in, int* out, size_t length) {
  for(size_t i{}; i < length; i++) {
    out[i] = fn(in[i]);
  }
}

int main() {
  constexpr size_t len{ 3 };
  int base[]{ 1, 2, 3 }, a[len], b[len], c[len];
  transform([](int x) { return 1; }, base, a, len);
  transform([](int x) { return x; }, base, b, len);
  transform([](int x) { return 10 * x + 5; }, base, c, len);
  for(size_t i{}; i < len; i++) {
    printf("Element %zd: %d %d %d\n", i, a[i], b[i], c[i]);
  }
}
```

Lambda Example

Don't be fooled. No different than `typename T`

```
#include <cstdint>
#include <cstdio>

template <typename Fn>
void transform(Fn fn, const int* in, int* out, size_t length) {
  for(size_t i{}; i < length; i++) {
    out[i] = fn(in[i]);
  }
}

int main() {
  constexpr size_t len{ 3 };
  int base[]{ 1, 2, 3 }, a[len], b[len], c[len];
  transform([](int x) { return 1; }, base, a, len);
  transform([](int x) { return x; }, base, b, len);
  transform([](int x) { return 10 * x + 5; }, base, c, len);
  for(size_t i{}; i < len; i++) {
    printf("Element %zd: %d %d %d\n", i, a[i], b[i], c[i]);
  }
}
```

Lambda Example

```
#include <cstdint>
#include <cstdio>

template <typename Fn>
void transform(Fn fn, const int* in, int* out, size_t length) {
  for(size_t i{}; i < length; i++) {
    out[i] = fn(in[i]);
  }
}

int main() {
  constexpr size_t len{ 3 };
  int base[]{ 1, 2, 3 }, a[len], b[len], c[len];
  transform([](int x) { return 1; }, base, a, len);
  transform([](int x) { return x; }, base, b, len);
  transform([](int x) { return 10 * x + 5; }, base, c, len);
  for(size_t i{}; i < len; i++) {
    printf("Element %zd: %d %d %d\n", i, a[i], b[i], c[i]);
  }
}
```

Don't be fooled. No different than `typename T`

Except you better provide a type that can be invoked, because of how it's used

Lambda Example

```
#include <cstdint>
#include <cstdio>

template <typename Fn>
void transform(Fn fn, const int* in, int* out, size_t length) {
  for(size_t i{}; i < length; i++) {
    out[i] = fn(in[i]);
  }
}

int main() {
  constexpr size_t len{ 3 };
  int base[]{ 1, 2, 3 }, a[len], b[len], c[len];
  transform([](int x) { return 1; }, base, a, len);
  transform([](int x) { return x; }, base, b, len);
  transform([](int x) { return 10 * x + 5; }, base, c, len);
  for(size_t i{}; i < len; i++) {
    printf("Element %zd: %d %d %d\n", i, a[i], b[i], c[i]);
  }
}
```

Output:

```
Element 0: 1 1 15
Element 1: 1 2 25
Element 2: 1 3 35
```

Lambda Example

```
#include <cstdint>
#include <cstdio>

template <typename Fn>
void transform(Fn fn, const int* in, int* out, size_t length) {
  for(size_t i{}; i < length; i++) {
    out[i] = fn(in[i]);
  }
}

int main() {
  constexpr size_t len{ 3 };
  int base[]{ 1, 2, 3 }, a[len], b[len], c[len];
  transform([](int x) { return 1; }, base, a, len);
  transform([](int x) { return x; }, base, b, len);
  transform([](int x) { return 10 * x + 5; }, base, c, len);
  for(size_t i{}; i < len; i++) {
    printf("Element %zd: %d %d %d\n", i, a[i], b[i], c[i]);
  }
}
```

Output:

```
Element 0: 1 1 15
Element 1: 1 2 25
Element 2: 1 3 35
```

Note that by declaring `transform` as a template function, you can reuse it with any function object.

Generic Lambdas

- *Generic lambdas* are lambda expression templates
  - For one or more parameter one specifies `auto` rather than a concrete type
  - the `auto` types becomes template parameters
    - Compiler will build a custom instantiation of the lambda

Generic Lambdas

```
#include <cstdint>
#include <cstdio>

template <typename Fn, typename T>
void transform(Fn fn, const T* in, T* out, size_t len) {
  for(size_t i{}; i < len; i++) {
    out[i] = fn(in[i]);
  }
}

int main() {
  constexpr size_t l{ 3 };
  int base_int[]{ 1, 2, 3 }, a[l];
  float base_float[]{ 10.f, 20.f, 30.f }, b[l];
  auto translate = [](auto x) { return 10 * x + 5; };
  transform(translate, base_int, a, l);
  transform(translate, base_float, b, l);

  for(size_t i{}; i < l; i++) {
    printf("Element %zd: %d %f\n", i, a[i], b[i]);
  }
}
```

You better provide types `Fn` and `T` such that `Fn` that can be invoked on objects of type `T`

Generic Lambdas

```
#include <cstdint>
#include <cstdio>

template <typename Fn, typename T>
void transform(Fn fn, const T* in, T* out, size_t len) {
  for(size_t i{}; i < len; i++) {
    out[i] = fn(in[i]);
  }
}

int main() {
  constexpr size_t l{ 3 };
  int base_int[]{ 1, 2, 3 }, a[l];
  float base_float[]{ 10.f, 20.f, 30.f }, b[l];
  auto translate = [](auto x) { return 10 * x + 5; };
  transform(translate, base_int, a, l);
  transform(translate, base_float, b, l);

  for(size_t i{}; i < l; i++) {
    printf("Element %zd: %d %f\n", i, a[i], b[i]);
  }
}
```

generic lambda

Generic Lambdas

```
#include <cstdint>
#include <cstdio>

template <typename Fn, typename T>
void transform(Fn fn, const T* in, T* out, size_t len) {
  for(size_t i{}; i < len; i++) {
    out[i] = fn(in[i]);
  }
}

int main() {
  constexpr size_t l{ 3 };
  int base_int[]{ 1, 2, 3 }, a[l];
  float base_float[]{ 10.f, 20.f, 30.f }, b[l];
  auto translate = [](auto x) { return 10 * x + 5; };
  transform(translate, base_int, a, l);
  transform(translate, base_float, b, l);

  for(size_t i{}; i < l; i++) {
    printf("Element %zd: %d %f\n", i, a[i], b[i]);
  }
}
```

Output:

```
Element 0: 15 105.000000
Element 1: 25 205.000000
Element 2: 35 305.000000
```

Lambda Captures

- Lambda captures inject objects into the lambda
  - This can be used to modify behavior of the lambda
  - Declared within brackets [ ]
  - Capture list before parameter list
  - Can contain any number of comma separated values
    - Which can then be used within lambda’s body
  - Can capture by reference or value

Lambda Captures

lambda version of `CountIf`

```
#include <cstdint>
#include <cstdio>

int main() {
  char to_count{ 's' };
  auto s_counter = [to_count](const char* str) {
    size_t index{}, result{};
    while(str[index]) {
      if(str[index] == to_count)
        result++;
      index++;
    }
    return result;
  };
  auto sally = s_counter("Sally sells seashells by the seashore.");
  printf("Sally: %zd\n", sally);
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zd\n", sailor);
}
```

`to_count` captured and can now be used within lambda's body

Lambda Captures

```
#include <cstdint>
#include <cstdio>

int main() {
  char to_count{ 's' };
  auto s_counter = [to_count](const char* str) {
    size_t index{}, result{};
    while(str[index]) {
      if(str[index] == to_count)
        result++;
      index++;
    }
    return result;
  };
  auto sally = s_counter("Sally sells seashells by the seashore.");
  printf("Sally: %zd\n", sally);
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zd\n", sailor);
}
```

Output:

```
Sally: 7
Sailor: 3
```

Lambda Captures

```
#include <cstdint>
#include <cstdio>

int main() {
  char to_count{ 's' };
  size_t tally{};
  auto s_counter = [to_count, &tally](const char* str) {
    size_t index{}, result{};
    while(str[index]) {
      if(str[index] == to_count)
        result++;
      index++;
    }
    tally += result;
    return result;
  };
  printf("Tally: %zd\n", tally);
  auto sally = s_counter("Sally sells seashells by the seashore.");
  printf("Sally: %zd\n", sally);
  printf("Tally: %zd\n", tally);
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zd\n", sailor);
  printf("Tally: %zd\n", tally);
}
```

Capture by reference

Note we are not declaring these so no need for type

Lambda Captures

```
#include <cstdint>
#include <cstdio>

int main() {
  char to_count{ 's' };
  size_t tally{};
  auto s_counter = [to_count, &tally](const char* str) {
    size_t index{}, result{};
    while(str[index]) {
      if(str[index] == to_count)
        result++;
      index++;
    }
    tally += result;
    return result;
  };
  printf("Tally: %zd\n", tally);
  auto sally = s_counter("Sally sells seashells by the seashore.");
  printf("Sally: %zd\n", sally);
  printf("Tally: %zd\n", tally);
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zd\n", sailor);
  printf("Tally: %zd\n", tally);
}
```

Output:

```
Tally: 0
Sally: 7
Tally: 7
Sailor: 3
Tally: 10
```

Recall: Function Pointers

- Declaring a function pointer is similar to declaring a function

```
#include <stdio.h>
void my_int_func(int x) {
    printf( "%d\n", x );
}


int main() {
    void (*foo)(int);
    /* the ampersand is actually optional */
    foo = &my_int_func;

    /* call my_int_func (note that you do not need to write (*foo)(2) ) */
    foo( 2 );
    /* but if you want to, you may */
    (*foo)( 2 );

    return 0;
}
```

Thanks Alex Allain:
https://www.cprogramming.com/tutorial/function-pointers.html

Recall: Function Pointers

- Declaring a function pointer is similar to declaring a function

```
#include <stdio.h>

double my_other_example(int a, int b, char* c) {

  return 0;

}


int main() {

    double (*my_func_ptr)(int, int, char*);
    my_func_ptr = my_other_example;

    return 0;
}
```

Aside: std::function

- `std::function` from `<functional>` header is a polymorphic container for callable objects
- In other words, a generic function pointer
  - You can store a static function, a function object, or a lambda into a `std::function`

Declaring a `function`

- To declare a `function` you must provide a single template parameter containing the function prototype of the callable object

```
std::function<return-type(arg-type-1, arg-type-2, etc.)>
```

- `std::function` class template has many constructors
  - Default constructor constructs a `std::function` in empty mode – it contains no callable object

Empty Functions

- If you declare a `std::function` with no contained object, "calling it" will throw a `std::bad_function_call` exception

```
#include <cstdio>
#include <functional>

int main() {
  std::function<void()> func;
  try {
    func();
  } catch(const std::bad_function_call& e) {
    printf("Exception: %s", e.what());
  }
}
```

```
Exception: std::exception
```

Assigning a Callable Object to a Function

- Two ways: use the constructor or use the assignment operator of `function`

```
#include <cstdio>
#include <functional>

void static_func() {
  printf("A static function.\n");
}

int main() {
  std::function<void()> func{ [] { printf("A lambda.\n"); } };
  func();
  func = static_func;
  func();
}
```

```
A lambda.
A static function.
```

Example

- You can construct a `function` with any callable object that supports the function semantics implied by the template parameter of the `function`

```cpp
#include <cstdint>
#include <cstdio>
#include <functional>

struct CountIf {
  --snip--
};

size_t count_spaces(const char* str) {
  size_t index{}, result{};
  while(str[index]) {
    if(str[index] == ' ')
      result++;
    index++;
  }
  return result;
}

std::function<size_t(const char*)> funcs[]{
  count_spaces,
  CountIf{ 'e' },
  [](const char* str) {
    size_t index{};
    while(str[index])
      index++;
    return index;
  }
};

auto text = "Sailor went to sea to see what he could see.";

int main() {
  size_t index{};
  for(const auto& func : funcs) {
    printf("func #%zd: %zd\n", index++, func(text));
  }
}
```

An array of `std::function` objects

```cpp
#include <cstdint>
#include <cstdio>
#include <functional>

struct CountIf {
  --snip--
};

size_t count_spaces(const char* str) {
  size_t index{}, result{};
  while(str[index]) {
    if(str[index] == ' ')
      result++;
    index++;
  }
  return result;
}

std::function<size_t(const char*)> funcs[]{
  count_spaces,
  CountIf{ 'e' },
  [](const char* str) {
    size_t index{};
    while(str[index])
      index++;
    return index;
  }
};

auto text = "Sailor went to sea to see what he could see.";

int main() {
  size_t index{};
  for(const auto& func : funcs) {
    printf("func #%zd: %zd\n", index++, func(text));
  }
}
```

```
func #0: 9
func #1: 7
func #2: 44
```

Runtime Overhead

- Using a `function` comes with a runtime overhead cost
  - `function` might need to make a dynamic allocation to store callable object
  - Compiler has difficulty optimizing away `function` invocations, so often incur an indirect function call
    - Requires additional pointer dereferences

Indirect Function Call?

- Direct function call: function call is made with a fixed address in instruction
  - For those in CS 301, `jal` to fixed address that has been placed in the executable by the linker
- Indirect function call: function call is made with address of callee in a register
  - Register is previously loaded either with fixed address of function being called, or with a value fetched from somewhere else (e.g., memory or another register) where the function address has been stored

Indirect Function Call?

- Direct function call: will always call the same function
- Indirect function call: can call different functions, depending on what was loaded in register before call is made
  - The indirection requires extra effort

Variadic Functions

- *Variadic functions* take a variable number of arguments
  - E.g., `printf` – you provide format specifier and variable number of parameters
  - Variadic functions declared by placing … as the final parameter
  - On invocation, compiler matches supplied parameters against declared arguments. Remainder are represented by …

Variadic Functions

- *Variadic functions* take a variable number of arguments

```
int sum(size_t n, ...) {
```

- Extract individual arguments from variadic arguments via utility functions in the `<cstdarg>` header

Variadic Functions

Table 9-1: Utility Functions in the <cstdarg> Header

Function	Description
va_list	Used to declare a local variable representing the variadic arguments
va_start	Enables access to the variadic arguments
va_end	Used to end iteration over the variadic arguments
va_arg	Used to iterate over each element in the variadic arguments
va_copy	Makes a copy of the variadic arguments

Variadic Functions

```
#include <cstdarg>
#include <cstdint>
#include <cstdio>

int sum(size_t n, ...) {
  va_list args;
  va_start(args, n);
  int result{};
  while(n--) {
    auto next_element = va_arg(args, int);
    result += next_element;
  }
  va_end(args);
  return result;
}

int main() {
  printf("The answer is %d.", sum(6, 2, 4, 6, 8, 10, 12));
}
```

Variadic Functions

```
#include <cstdarg>
#include <cstdint>
#include <cstdio>

int sum(size_t n, ...) {
  va_list args;
  va_start(args, n);
  int result{};
  while(n--) {
    auto next_element = va_arg(args, int);
    result += next_element;
  }
  va_end(args);
  return result;
}

int main() {
  printf("The answer is %d.", sum(6, 2, 4, 6, 8, 10, 12));
}
```

All variadic functions must declare a `va_list`. Here it's called `args`

Variadic Functions

```
#include <cstdarg>
#include <cstdint>
#include <cstdio>

int sum(size_t n, ...) {
  va_list args;
  va_start(args, n);
  int result{};
  while(n--) {
    auto next_element = va_arg(args, int);
    result += next_element;
  }
  va_end(args);
  return result;
}

int main() {
  printf("The answer is %d.", sum(6, 2, 4, 6, 8, 10, 12));
}
```

All variadic functions must declare a `va_list`. Here it's called `args`