20.7 — Lambda captures

Capture clauses and capture by value

In the previous lesson (20.6 -- Introduction to lambdas (anonymous functions)), we introduced this example:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

int main()
{
  std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };

  auto found{ std::find_if(arr.begin(), arr.end(),
                           [](std::string_view str)
                           {
                             return str.find("nut") != std::string_view::npos;
                           }) };

  if (found == arr.end())
  {
    std::cout << "No nuts\n";
  }
  else
  {
    std::cout << "Found " << *found << '\n';
  }

  return 0;
}
```

Now, let’s modify the nut example and let the user pick a substring to search for. This isn’t as intuitive as you might expect.

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>
#include <string>

int main()
{
  std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };

  // Ask the user what to search for.
  std::cout << "search for: ";

  std::string search{};
  std::cin >> search;

  auto found{ std::find_if(arr.begin(), arr.end(), [](std::string_view str) {
    // Search for @search rather than "nut".
    return str.find(search) != std::string_view::npos; // Error: search not
accessible in this scope
  }) };

  if (found == arr.end())
  {
    std::cout << "Not found\n";
  }
  else
  {
    std::cout << "Found " << *found << '\n';
  }

  return 0;
}
```

This code won't compile. Unlike nested blocks, where any identifier defined in an outer block is accessible in the scope of the nested block, lambdas can only access specific kinds of identifiers: global identifiers, entities that are known at compile time, and entities with static storage duration. `search` fulfills none of these requirements, so the lambda can't see it. That's what the capture clause is there for.

The capture clause

The **capture clause** is used to (indirectly) give a lambda access to variables available in the surrounding scope that it normally would not have access to. All we need to do is list the entities we want to access from within the lambda as part of the capture clause. In this case, we want to give our lambda access to the value of variable `search`, so we add it to the capture clause:

```cpp
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>
#include <string>

int main()
{
  std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };

  std::cout << "search for: ";

  std::string search{};
  std::cin >> search;

  // Capture @search                                vvvvvv
  auto found{ std::find_if(arr.begin(), arr.end(), [search](std::string_view str) {
    return str.find(search) != std::string_view::npos;
  }) };

  if (found == arr.end())
  {
    std::cout << "Not found\n";
  }
  else
  {
    std::cout << "Found " << *found << '\n';
  }

  return 0;
}
```

The user can now search for an element of our array.

Output

```
search for: nana
Found banana
```

So how do captures actually work?

While it might look like our lambda in the example above is directly accessing the value of `main`'s `search` variable, this is not the case. Lambdas might look like nested blocks, but they work slightly differently (and the distinction is important).

When a lambda definition is executed, for each variable that the lambda captures, a clone of that variable is made (with an identical name) inside the lambda. These cloned variables are initialized from the outer scope variables of the same name at this point.

Thus, in the above example, when the lambda object is created, the lambda gets its own cloned variable named `search`. This cloned `search` has the same value as `main`'s `search`, so it behaves like we're accessing `main`'s `search`, but we're not.

While these cloned variables have the same name, they don't necessarily have the same type as the original variable. We'll explore this in the upcoming sections of this lesson.

Key insight

The captured variables of a lambda are *copies* of the outer scope variables, not the actual variables.

For advanced readers

Although lambdas look like functions, they're actually objects that can be called like functions (these are called **functors** -- we'll discuss how to create your own functors from scratch in a future lesson).

When the compiler encounters a lambda definition, it creates a custom object definition for the lambda. Each captured variable becomes a data member of the object.

At runtime, when the lambda definition is encountered, the lambda object is instantiated, and the members of the lambda are initialized at that point.

Captures are treated as const by default

When a lambda is called, `operator()` is invoked. By default, this `operator()` treats captures as const, meaning the lambda is not allowed to modify those captures.

In the following example, we capture the variable `ammo` and try to decrement it.

```
#include <iostream>

int main()
{
  int ammo{ 10 };

  // Define a lambda and store it in a variable called "shoot".
  auto shoot{
    [ammo]() {
      // Illegal, ammo cannot be modified.
      --ammo;

      std::cout << "Pew! " << ammo << " shot(s) left.\n";
    }
  };

  // Call the lambda
  shoot();

  std::cout << ammo << " shot(s) left\n";

  return 0;
}
```

The above won’t compile, because `ammo` is treated as const within the lambda.

Mutable captures

To allow modifications of variables that were captured, we can mark the lambda as `mutable`:

```
#include <iostream>

int main()
{
  int ammo{ 10 };

  auto shoot{
    [ammo]() mutable { // now mutable
      // We're allowed to modify ammo now
      --ammo;

      std::cout << "Pew! " << ammo << " shot(s) left.\n";
    }
  };

  shoot();
  shoot();

  std::cout << ammo << " shot(s) left\n";

  return 0;
}
```

Output:

```
Pew! 9 shot(s) left.
Pew! 8 shot(s) left.
10 shot(s) left
```

While this now compiles, there’s still a logic error. What happened? When the lambda was called, the lambda captured a *copy* of `ammo`. When the lambda decremented `ammo` from `10` to `9` to `8`, it decremented its own copy, not the original `ammo` value in `main()`.

Note that the value of `ammo` is preserved across calls to the lambda!

Warning

Because captured variables are members of the lambda object, their values are persisted across multiple calls to the lambda!

Capture by reference

Much like functions can change the value of arguments passed by reference, we can also capture variables by reference to allow our lambda to affect the value of the argument.

To capture a variable by reference, we prepend an ampersand (`&`) to the variable name in the capture. Unlike variables that are captured by value, variables that are captured by reference are non-const, unless the variable they’re capturing is `const`. Capture by reference should be preferred over capture by value whenever you would normally prefer passing an argument to a function by reference (e.g. for non-fundamental types).

Here’s the above code with `ammo` captured by reference:

```
#include <iostream>

int main()
{
  int ammo{ 10 };

  auto shoot{
    // We don't need mutable anymore
    [&ammo]() { // &ammo means ammo is captured by reference
      // Changes to ammo will affect main's ammo
      --ammo;

      std::cout << "Pew! " << ammo << " shot(s) left.\n";
    }
  };

  shoot();

  std::cout << ammo << " shot(s) left\n";

  return 0;
}
```

This produces the expected answer:

```
Pew! 9 shot(s) left.
9 shot(s) left
```

Now, let’s use a reference capture to count how many comparisons `std::sort` makes when it sorts an array.

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

struct Car
{
  std::string_view make{};
  std::string_view model{};
};

int main()
{
  std::array<Car, 3> cars{ { { "Volkswagen", "Golf" },
                             { "Toyota", "Corolla" },
                             { "Honda", "Civic" } } };

  int comparisons{ 0 };

  std::sort(cars.begin(), cars.end(),
    // Capture @comparisons by reference.
    [&comparisons](const auto& a, const auto& b) {
      // We captured comparisons by reference. We can modify it without "mutable".
      ++comparisons;

      // Sort the cars by their make.
      return a.make < b.make;
  });

  std::cout << "Comparisons: " << comparisons << '\n';

  for (const auto& car : cars)
  {
    std::cout << car.make << ' ' << car.model << '\n';
  }

  return 0;
}
```

Possible output

```
Comparisons: 2
Honda Civic
Toyota Corolla
Volkswagen Golf
```

Capturing multiple variables

Multiple variables can be captured by separating them with a comma. This can include a mix of variables captured by value or by reference:

```
int health{ 33 };
int armor{ 100 };
std::vector<CEnemy> enemies{};

// Capture health and armor by value, and enemies by reference.
[health, armor, &enemies](){};
```

Default captures

Having to explicitly list the variables you want to capture can be burdensome. If you modify your lambda, you may forget to add or remove captured variables. Fortunately, we can enlist the compiler's help to auto-generate a list of variables we need to capture.

A **default capture** (also called a **capture-default**) captures all variables that are mentioned in the lambda. Variables not mentioned in the lambda are not captured if a default capture is used.

To capture all used variables by value, use a capture value of `=`.
To capture all used variables by reference, use a capture value of `&`.

Here's an example of using a default capture by value:

```
#include <algorithm>
#include <array>
#include <iostream>

int main()
{
  std::array areas{ 100, 25, 121, 40, 56 };

  int width{};
  int height{};

  std::cout << "Enter width and height: ";
  std::cin >> width >> height;

  auto found{ std::find_if(areas.begin(), areas.end(),
                           [=](int knownArea) { // will default capture width and
height by value
                             return width * height == knownArea; // because they're
mentioned here
                           }) };

  if (found == areas.end())
  {
    std::cout << "I don't know this area :(\n";
  }
  else
  {
    std::cout << "Area found :)\n";
  }

  return 0;
}
```

Default captures can be mixed with normal captures. We can capture some variables by value and others by reference, but each variable can only be captured once.

```
int health{ 33 };
int armor{ 100 };
std::vector<CEnemy> enemies{};

// Capture health and armor by value, and enemies by reference.
[health, armor, &enemies](){};

// Capture enemies by reference and everything else by value.
[=, &enemies](){};

// Capture armor by value and everything else by reference.
[&, armor](){};

// Illegal, we already said we want to capture everything by reference.
[&, &armor](){};

// Illegal, we already said we want to capture everything by value.
[=, armor](){};

// Illegal, armor appears twice.
[armor, &health, &armor](){};

// Illegal, the default capture has to be the first element in the capture group.
[armor, &](){};
```

Defining new variables in the lambda-capture

Sometimes we want to capture a variable with a slight modification or declare a new variable that is only visible in the scope of the lambda. We can do so by defining a variable in the lambda-capture without specifying its type.

```
#include <array>
#include <iostream>
#include <algorithm>

int main()
{
  std::array areas{ 100, 25, 121, 40, 56 };

  int width{};
  int height{};

  std::cout << "Enter width and height: ";
  std::cin >> width >> height;

  // We store areas, but the user entered width and height.
  // We need to calculate the area before we can search for it.
  auto found{ std::find_if(areas.begin(), areas.end(),
                           // Declare a new variable that's visible only to the
lambda.
                           // The type of userArea is automatically deduced to int.
                           [userArea{ width * height }](int knownArea) {
                             return userArea == knownArea;
                           }) };

  if (found == areas.end())
  {
    std::cout << "I don't know this area :(\n";
  }
  else
  {
    std::cout << "Area found :)\n";
  }

  return 0;
}
```

`userArea` will only be calculated once when the lambda is defined. The calculated area is stored in the lambda object and is the same for every call. If a lambda is mutable and modifies a variable that was defined in the capture, the original value will be overridden.

Best practice

Only initialize variables in the capture if their value is short and their type is obvious. Otherwise it's best to define the variable outside of the lambda and capture it.

Dangling captured variables

Variables are captured at the point where the lambda is defined. If a variable captured by reference dies before the lambda, the lambda will be left holding a dangling reference.

For example:

```
#include <iostream>
#include <string>

// returns a lambda
auto makeWalrus(const std::string& name)
{
  // Capture name by reference and return the lambda.
  return [&]() {
    std::cout << "I am a walrus, my name is " << name << '\n'; // Undefined behavior
  };
}

int main()
{
  // Create a new walrus whose name is Roofus.
  // sayName is the lambda returned by makeWalrus.
  auto sayName{ makeWalrus("Roofus") };

  // Call the lambda function that makeWalrus returned.
  sayName();

  return 0;
}
```

The call to `makeWalrus` creates a temporary `std::string` from the string literal “Roofus”. The lambda in `makeWalrus` captures the temporary string by reference. The temporary string dies when `makeWalrus` returns, but the lambda still references it. Then when we call `sayName`, the dangling reference is accessed, causing undefined behavior.

Note that this also happens if `name` is passed to `makeWalrus` by value. The variable `name` still dies at the end of `makeWalrus`, and the lambda is left holding a dangling reference.

Warning

Be extra careful when you capture variables by reference, especially with a default reference capture. The captured variables must outlive the lambda.

If we want the captured `name` to be valid when the lambda is used, we need to capture it by value instead (either explicitly or using a default-capture by value).

Unintended copies of mutable lambdas

Because lambdas are objects, they can be copied. In some cases, this can cause problems. Consider the following code:

```
#include <iostream>

int main()
{
  int i{ 0 };

  // Create a new lambda named count
  auto count{ [i]() mutable {
    std::cout << ++i << '\n';
  } };

  count(); // invoke count

  auto otherCount{ count }; // create a copy of count

  // invoke both count and the copy
  count();
  otherCount();

  return 0;
}
```

Output

```
1
2
2
```

Rather than printing 1, 2, 3, the code prints 2 twice. When we created `otherCount` as a copy of `count`, we created a copy of `count` in its current state. `count`'s `i` was 1, so `otherCount`'s `i` is 1 as well. Since `otherCount` is a copy of `count`, they each have their own `i`.

Now let's take a look at a slightly less obvious example:

```
#include <iostream>
#include <functional>

void myInvoke(const std::function<void()>& fn)
{
    fn();
}

int main()
{
    int i{ 0 };

    // Increments and prints its local copy of @i.
    auto count{ [i]() mutable {
      std::cout << ++i << '\n';
    } };

    myInvoke(count);
    myInvoke(count);
    myInvoke(count);

    return 0;
}
```

Output:

```
1
1
1
```

This exhibits the same problem as the prior example in a more obscure form. When `myInvoke()` is called, the `std::function` parameter is initialized with our lambda, and internally makes a copy of the lambda object. Thus, our call to `fn()` is actually being executed on the copy of our lambda, not the actual lambda.

If we need to pass a mutable lambda, and want to avoid the possibility of inadvertent copies being made, there are two options. One option is to use a non-capturing lambda instead -- in the above case, we could remove the capture and track our state using a static local variable instead. But static local variables can be difficult to keep track of and make our code less readable. A better option is to prevent copies of our lambda from being made in the first place. But since we can't affect how `std::function` (or other standard library functions or objects) are implemented, how can we do this?

One option (h/t to reader Dck) is to put our lambda into a `std::function` immediately. That way, when we call `myInvoke()`, the reference parameter `fn` can bind to our `std::function`, and no temporary copy is made:

```
#include <iostream>
#include <functional>

void myInvoke(const std::function<void()>& fn)
{
    fn();
}

int main()
{
    int i{ 0 };

    // Increments and prints its local copy of @i.
    std::function count{ [i]() mutable { // lambda object stored in a std::function
      std::cout << ++i << '\n';
    } };

    myInvoke(count); // doesn't create copy when called
    myInvoke(count); // doesn't create copy when called
    myInvoke(count); // doesn't create copy when called

    return 0;
}
```

Our output is now as expected:

```
1
2
3
```

An alternate solution is to use a reference wrapper. C++ provides a convenient type (as part of the <functional> header) called `std::reference_wrapper` that allows us to pass a normal type as if it was a reference. For even more convenience, a `std::reference_wrapper` can be created by using the `std::ref()` function. By wrapping our lambda in a `std::reference_wrapper`, whenever anybody tries to make a copy of our lambda, they'll make a copy of the reference_wrapper instead (avoiding making a copy of the lambda).

Here's our updated code using `std::ref`:

```
#include <iostream>
#include <functional> // includes std::reference_wrapper and std::ref

void myInvoke(const std::function<void()>& fn)
{
    fn();
}

int main()
{
    int i{ 0 };

    // Increments and prints its local copy of @i.
    auto count{ [i]() mutable {
      std::cout << ++i << '\n';
    } };

    // std::ref(count) ensures count is treated like a reference
    // thus, anything that tries to copy count will actually copy the reference
    // ensuring that only one count exists
    myInvoke(std::ref(count));
    myInvoke(std::ref(count));
    myInvoke(std::ref(count));

    return 0;
}
```

Our output is now as expected:

```
1
2
3
```

What's interesting about this method is that it works even if `myInvoke` takes `fn` by value (instead of by reference)!

Rule

Standard library functions may copy function objects (reminder: lambdas are function objects). If you want to provide lambdas with mutable captured variables, pass them by reference using `std::ref`.

Best practice

Try to avoid mutable lambdas. Non-mutable lambdas are easier to understand and don't suffer from the above issues, as well as more dangerous issues that arise when you add parallel execution.

Quiz time

Question #1

Which of the following variables can be used by the lambda in `main` without explicitly capturing them?

```
int i{};
static int j{};

int getValue()
{
  return 0;
}

int main()
{
  int a{};
  constexpr int b{};
  static int c{};
  static constexpr int d{};
  const int e{};
  const int f{ getValue() };
  static const int g{};
  static const int h{ getValue() };

  [](){
    // Try to use the variables without explicitly capturing them.
    a;
    b;
    c;
    d;
    e;
    f;
    g;
    h;
    i;
    j;
  }();

  return 0;
}
```

Show Solution

Question #2

What does the following code print? Don't run the code, work it out in your head.

```
#include <iostream>
#include <string>

int main()
{
  std::string favoriteFruit{ "grapes" };

  auto printFavoriteFruit{
    [=]() {
      std::cout << "I like " << favoriteFruit << '\n';
    }
  };

  favoriteFruit = "bananas with chocolate";

  printFavoriteFruit();

  return 0;
}
```

Show Solution

Question #3

We're going to write a little game with square numbers (numbers which can be created by multiplying an integer with itself (1, 4, 9, 16, 25, …)).

To setup the game:

- Ask the user to enter a number to start at (e.g. 3).
- Ask the user how many values to generate.
- Pick a random integer between 2 and 4. This is the multiplier.
- Generate the number of values the user indicated. Begining with the starting number, each value should be the next square number, multiplied by the multiplier.

To play the game:

- The user enters a guess.
- If the guess matches any generated value, the value is removed from the list and the user gets to guess again.
- If the user guesses all of the generated values, they win.
- If the guess does not match a generated value, the user loses and the program tells them the nearest unguessed value.

Here are a couple of sample sessions to give you a better understanding of how the game works:

```
Start where? 4
How many? 5
I generated 5 square numbers. Do you know what each number is after multiplying it by
2?
> 32
Nice! 7 number(s) left.
> 72
Nice! 6 number(s) left.
> 50
Nice! 5 number(s) left.
> 126
126 is wrong! Try 128 next time.
```

- Starting at 4, the program generates the next 5 squares: 16, 25, 36, 49, 64
- The program picked 2 as the random multiplier, so each square is multiplied by 2: 32, 50, 72, 98, 128
- Now the user gets to guess.
- 32 is in the list.
- 72 is in the list.
- 126 is not in the list, so the user loses. The closest unguessed number is 128.

```
Start where? 1
How many? 3
I generated 3 square numbers. Do you know what each number is after multiplying it by
4?
> 4
Nice! 2 numbers left.
> 16
Nice! 1 numbers left.
> 36
Nice! You found all numbers, good job!
```

- Starting at 1, the program generates the next 3 squares: 1, 4, 9
- The program picked 4 as the random multiplier, so each square is multiplied by 4: 4, 16, 36
- The user guesses all numbers correctly and wins the game.

Tips:
Use `std::find` (18.3 -- Introduction to standard library algorithms) to search for a number in the list.

Use `std::vector::erase` to remove an element, e.g.

```
auto found{ std::find(/* ... */) };

// Make sure the element was found

myVector.erase(found);
```

Use `std::min_element` and a lambda to find the number closest to the user’s guess. `std::min_element` works analogous to `std::max_element` from the previous quiz.

Show Hint

Show Solution