Class 06

More on Memory,

Lambdas,

Const

Outline

- Brief Introduction to Lambda Functions
- More on Memory
  - Variable Scope
  - Pass-by-Value
  - References
- Const Correctness

Lambda Functions

- Lambda functions are special functions in C++ that are defined *inline with your code*.
- When paired with algorithms we can create predicates: functions to execute with the algorithm.
- Lambdas are defined as follows:

```
auto my_lambda = [](T1 arg1, T2 arg2, ...) -> return_type
{
    // do stuff...
};
```

  - We *must* use auto to define a lambda (their types are extremely complex!).
  - Start the lambda with square braces (more on this in a few slides)
  - Next the arguments are specified like a normal function (arguments are optional)
  - Next the return type is specified (specifying the return type is but always do so!)
  - Lastly, the body of the lambda function is provided between curly braces, and ended with a semicolon

Lambda Functions

- Example

```
auto data = std::vector<int>{1, 2, 3, 4, 5, 6};
auto mod5_sorter = [](int a, int b) -> bool
{
    return (a % 5) < (b % 5);
};
std::ranges::sort(data, mod5_sorter);  // 5, 1, 6, 2, 3, 4
```

Lambda Functions

- Let's break it down:

```
auto mod5_sorter = [](int a, int b) -> bool
{
    return (a % 5) < (b % 5);
};
```

- Here we are creating a new **variable** that is a lambda!

- The name of the lambda variable is `mod5_sorter`
  - It takes in two integers as input
  - It returns a bool
  - This returns the comparison (a % 5) < (b % 5)

Lambda Functions

- Commonly we will use lambdas to easily define transforms:

```
auto data = std::vector<int>{1, 2, 3, 4, 5, 6};
auto mod5 = [](int a) -> int
{
    return a % 5;
};
std::ranges::transform(data, data.begin(), mod5);
```

- The name of the lambda variable is `mod5`
  - It takes in one integer as input
  - It returns an int. Note that this is denoted differently, as `-> int`
  - This returns the input number in modulus 5.

Lambda Functions

- The square braces at the beginning of the lambda is called the *capture group*.
- By default, lambdas do not inherit variables in the current scope unless they are explicitly given to the lambda.

```
auto n = int{5};
auto data = std::vector<int>{1, 2, 3, 4, 5, 6};
auto mod5 = [n](int a) -> int
{
    return a % n;   // we can use n in the lambda because we captured it!
};
std::ranges::transform(data, data.begin(), mod5);
```

Variable Scope

- Consider the following program:

```
#include <vector>
auto main() -> int {
    auto x = int{10};
    auto y = double{100.0};
    auto data = std::vector<double>{1.0, 2.0, 3.0, 4.0, 5.0, 6.0};
}
```

- Work is done to create each variable (allocate memory, store the value, etc.)
- Variables are not just created but they are also destroyed.
- Any memory we take from the system we must eventually give back – this is called <u>*destroying,*</u> or <u>*deleting,*</u> or <u>*deallocating*</u> the memory.
  - So, when does this happen?

Variable Scope

- The keys here are the curly braces { and }.
- We have referred to a pair of braces creating a *block* of code, and they have a much deeper meaning than simply just sectioning code.
- The curly braces are used to indicate the *scope* of variables.
- *Scope* is the designated lifetime of the variable and its accessibility.

Variable Scope

- Blocks can contain other blocks.
- Nested blocks have access to variables created in their parent blocks *above them*.
- Variables created in one scope are not available to other scopes that are not nested within.

Variable Scope

- Consider the following:

```
auto ready = true;
if (ready) {
    auto rate = double{1.0};
   // .. do more stuff
}
fmt::print("Rate of reaction: {}\n", rate);
```

- What is the problem here?

Variable Scope

- Consider the following:

```
auto ready = true;
if (ready) {
    auto rate = double{1.0};
   // .. do more stuff
}
fmt::print("Rate of reaction: {}\n", rate);
```

- What happens if by chance that the variable *ready* is false? Then the variable *rate* would not exist and our statement to print it would be invalid!

- The variable *rate* is created *and destroyed* in the if-block. Once we leave that block, it is no longer accessible. *Scope* prevents the non-existence issue!

Variable Scope

- Consider the following:

```
{
    auto a = int{10};
    fmt::print("{}\n", a);
}
{
    int a = {100};
    fmt::print("{}\n", a);
}
fmt::print("{}\n", a);
```

- What is the problem here?

Variable Scope

- Consider the following:

```
{
    auto a = int{10};
    fmt::print("{}\n", a);
}
{
    int a = {100};
    fmt::print("{}\n", a);
}
fmt::print("{}\n", a);
```

- While the cout statements inside of each block are valid, the final one is not! Neither variable *a* exists anymore by the time we get to the final cout.

- It is totally valid to have variable names that overlap if they are in different scopes, this is not an issue!

Variable Scope

- Any time you see a new set of curly braces in the following, a new block of scope is created.
  - if-blocks
  - while loops
  - for loops
    - This new scope encapsulates any variables defined in the initial setup for the loop.
  - functions
    - ***This new scope encapsulates arguments to functions - this means that function arguments are defined within the function scope!***
    - ***This means that function arguments are <u>copies</u> of the data passed into the function!***

- When the closing brace of a *block* is encountered, all variables created within that scope are *destroyed*.

Pass-by-Value

- Consider the following:

```
auto swap(double a, double b) -> void {
   auto temp = a;
   a = b;
   b = temp;
}

// some other stuff...
auto x = double{10.0};
auto y = double{100.0};
swap(x, y);
```

- What is the problem here?

Pass-by-Value

- Consider the following:

```
auto swap(double a, double b) -> void {
   auto temp = a;
   a = b;
   b = temp;
}

// some other stuff...
auto x = double{10.0};
auto y = double{100.0};
swap(x, y);
```

- All variables in our function *swap* are copies of whatever we passed into it!

- This means that we are no longer working with x and y, but copies of x and y. Therefore, the copies are what are changed, and then go out of scope!

Pass-by-Value

- When you pass variables into a function, you are not passing the variable, but a *copy of the variable instead.*

- This means that calling a function has *overhead*.
  - e.g. calling a function with 8 parameters (say 4 ints and 4 doubles) means that we require 48 bytes of memory just to call the function!
  - Copying the inputs means memory needs to be allocated, assigned values, and then destroyed when the function ends.

- While 48 bytes is essentially nothing, imagine copying a vector that contains many thousands of elements where each element is 512 bytes! Such a vector with 10k elements can easily be 5MB. Before we address this, we need to talk about another concept.

References

- A reference in C++ is very similar to the standard definition of a reference:
  - "the action of mentioning or alluding to something."
- In C++ we can create *references* to variables. These references are like aliases:
  - If A is a reference to B, they are the same variable, just with a different name.
  - e.g. The God of Thunder is a reference to Thor; they are the same being.
- As references are just pointing to the variable it is referencing, it naturally shares the same memory address!
- Functions can take references as inputs, and even return references.

References

- How do we create references?

```cpp
auto x = int{0};
auto y = x;
auto &z = x; // z is a reference to x

// prints 0 0 0
fmt::print("{0} {1} {2}\n", x, y, z);

x = 1;

// prints 1 0 1
fmt::print("{0} {1} {2}\n", x, y, z);
```

References

- When we create a reference, we do so by placing an ampersand between the data type and the variable name; the spacing does not matter.
  - e.g. here y, z, and q are all references of x

```
auto x = int{1};
auto& y = x;
auto &z = x;
auto   &   q = x; // this is bad though...
```

  - Note that we do not change what is on the *right* of the assignment operator!

References

0x014	0x015	0x016	0x017	0x018	0x019	0x01A	0x01B	0x01C	0x01D	0x01E	0x01F
x = 1											

- Consider the following:

```
auto x = int{1};
auto &z = x;
```

- Where is z in our memory table?

References

0x014	0x015	0x016	0x017	0x018	0x019	0x01A	0x01B	0x01C	0x01D	0x01E	0x01F
x (a.k.a. "z") = 1											

- Consider the following:

```
auto x = int{1};
auto &z = x;
```

- Compilers will treat it like this above – z is a true alias and is in fact x through and through.

References

- Here are some rules about how references behave and how they are used:
  - You must initialize a reference with another variable: references **cannot be empty.**
  - References cannot be reassigned new variables.
  - When a reference goes out of scope, the original variable is unaffected.
    - e.g. if Z references X and Z is destroyed, you can still access X
  - When the variable being referenced is destroyed, if the reference persists, then we say that the reference **dangles.**
    - Accessing a dangling reference is <u>undefined behavior</u> and would likely cause your program to crash!

References

THIS SLIDE IS IMPORTANT. C++ IS CONFUSING AND THE USAGE AND MEANING OF AMPERSANDS WILL CONFUSE YOU.

- Consider the following:

```
#include <fmt/format.h>
auto main() -> int {
    auto x = int{10};
    auto &y = x;  // make a reference to x

    fmt::print("{}\n", x);
    fmt::print("{}\n", y);
    fmt::print("{}\n", &x);  // get the memory address of x
    fmt::print("{}\n", &y);  // get the memory address of y
}
```

- <u>When the & is with the variable definition</u>, it signals that the variable is a reference to another
- <u>When the & precedes a variable name</u>, it signals to grab the memory address of the variable!

Pass-by-Reference

- Consider this new swap function:

```
auto swap(double &a, double &b) -> void {
   auto temp = a;
   a = b;
   b = temp;
}
// some other stuff...
auto x = double{10.0};
auto y = double{100.0};
swap(x, y);
```

- As we are passing references to doubles, not just doubles, this means that we are not passing copies of those variables, but the actual variables themselves!

- Anything that happens to the references within the function is modifying the original variable.

Improper Use of References

- What's wrong here?

```
#include <fmt/format.h>

auto get_new_variable() -> int& {
    auto a = int{10};
    return a;
}

auto main() -> int {
    auto &x = get_new_variable();
    fmt::print("{}\n", x);
}
```

Improper Use of References

```cpp
#include <fmt/format.h>

auto get_new_variable() -> int& {
    auto a = int{10};
    return a;
}

auto main() -> int {
    auto &x = get_new_variable();
    fmt::print("{}\n", x);
}
```

- This program will exhibit *undefined behavior*. The variable *a* inside of our function goes out of scope, and thus the reference that is returned is dangling!

Improper Use of References

- What's wrong here?

```
#include <vector>

auto add_elements(std::vector<int> data) -> void {
    data.reserve(100);
    for (auto i = int{0}; i < 100; ++i) {
        data.push_back(i);
    }
}

auto main() -> int {
    auto d = std::vector<int>{};
    add_elements(d);
}
```

Improper Use of References

```
#include <vector>

auto add_elements(std::vector<int> data) -> void {
    data.reserve(100);
    for (auto i = int{0}; i < 100; ++i) {
        data.push_back(i);
    }
}

auto main() -> int {
    auto d = std::vector<int>{};
    add_elements(d);
}
```

- The vector is passed by value!
- The push_backs are happening on a copy of the vector, and it will all be lost when the function ends!

Pass-by-Reference

- Functions can *return references*, so long as the reference being returned *was passed into the function as an input*.
- Returning a reference to something created inside of the function will go out of scope, and when it does any reference to it will become dangling.
- Returning references to what was passed in allows us to create function/transformation chains.
  - In basic code this is overkill, but something we can employ later to great effect.

Pass-by-Reference

```cpp
#include <fmt/format.h>

auto square(int &x) -> int& {
    x *= x;
    return x;
}

auto add_two(int &x) -> int& {
    x += 2;
    return x;
}

auto main() -> int {
    int x = 10;
    add_two(square(x));  // first we square x, then we add 2 to it
    fmt::print("{}\n", x);
}
```

Pass-by-Reference

- Sometimes we do not want to modify a variable passed into a function, but we also do not want to copy it either.
- Copying can be expensive (think about copying an extremely large vector!), and this alone can be reason enough to pass something by reference.
- However, if something is passed by reference *and we do not want to modify it*, then we need a way to indicate that the variable is not to be touched!

Const-Correctness

- We can label variables (including function parameters and return types) as being *constant* using the keyword *const*.
- A variable that is *const* cannot be changed.
  - There are ways to do this, but ultimately making something non-const when it originally was const is undefined behavior.

```
#include <fmt/format.h>
auto main() -> int {
    const auto x = int{10};
    fmt::print("{}\n", x);
}
```

- Making a variable const is as simple as just adding the keyword before the type during its definition

Const-Correctness

- *Const-correctness* refers to ensuring that data that should be immutable *is* immutable.

- Modern compilers can use the fact that certain data is *const* to utilize optimizations into your compiled code.

- Some even advocate that by default we should make *every* variable *const* until we know that it will need to change.
  - *We will not be doing this.*

Const-Correctness, References, & Functions

- We are going to abide by the following rules:
  - Parameters to functions will be *const* if they do not change or need to change.
  - Non-primitive types will be by default be passed as const-references.
  - Non-primitive types will be passed-by-value if we need a copy to modify.

Const-Correctness, References, & Functions

```cpp
// some function signatures to illustrate what we want to aim for

// parameters are const to indicate that these values cannot change when being used in computations
auto compute_displacement(const double time, const double velocity, const double acceleration) -> double;

// data is a references to avoid a copy, and is const because we do not need to change the vector to
// compute the average
auto compute_average(const std::vector<double> &data) -> double;


// angles_in_rads is a reference but is not const as we are going to modify the vector directly
auto convert_angles_to_deg(std::vector<double> &angles_in_rads) -> void;


// angles_in_rads is neither const nor a reference as we are going to modify the vector but do not
// want to change the original vector
auto log_angles_in_deg(std::vector<double> angles_in_rads) -> void;
```