Part 2 - Basics

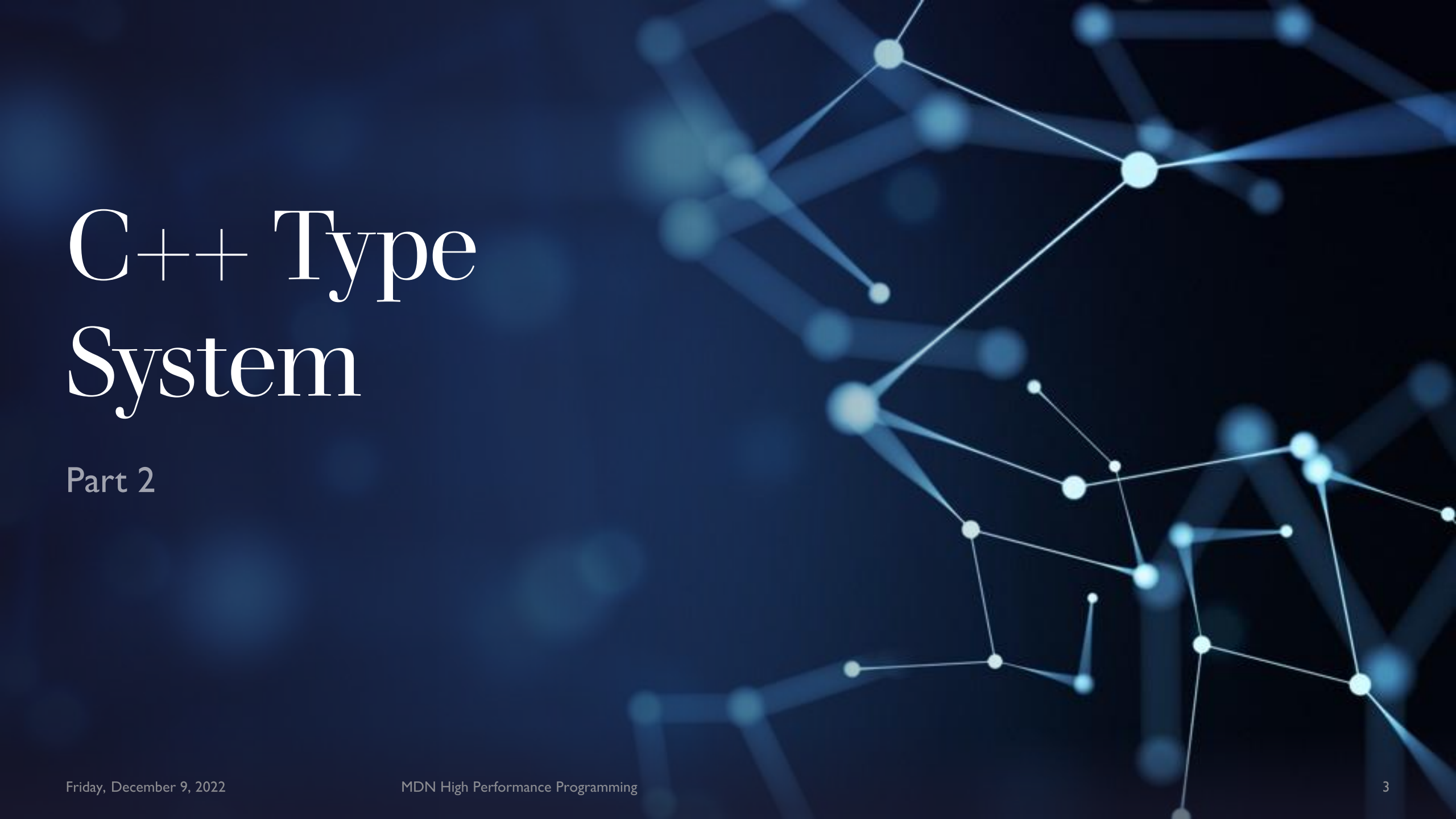
Tyler Swann

Agenda

- Recap
- C++ Type System
- Types
- Variables
- Operators
- IO
- Conditionals
- Loops
- Functions
- Discussion

C++ Type System

Part 2

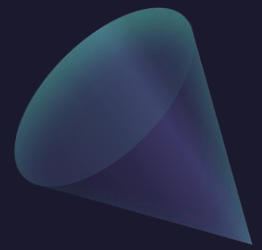


Type System

WHAT IS A TYPE SYSTEM

- A type system is a set of rules that govern the behavior and form the basis of the grammar of a language.
- How a programming languages dictate the notation if types and how types are assumed form the basis of its type system.

C++



- C++ has a strong type system
- C++ is statically typed
- C++ has a very rigorous definition of its type system and the various relationship between types
- C++ has the following type categories
  - Literals
  - Values
  - Types
  - Typeclasses

Types

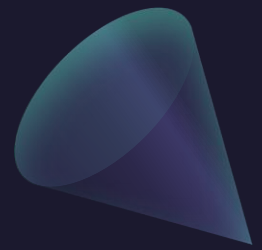
Part 2

Integral and Floating-Point Types

- `bool` – Boolean type – 8-bits – 1-byte
- `char` – character type – 8-bits – 1-byte
- `wchar_t` – wide character type – 16-bits or 32-bits – 2-bytes or 4-bytes
- `int` – integer type – 32-bits – 4-bytes
- `float` – single precision, floating-point type – binary32 format
- `double` – double precision, floating-point type – binary64 format

Other Types

- `void` – incomplete type – denotes no return.
- `nullptr` – literal for a pointer to nothing
- `std::nullptr_t` – type of `nullptr`
- `std::size_t` – Platform specific, maximum unsigned integer value
- `std::ptrdiff_t` – Type returned by the subtraction of two pointers
- `auto` – Automatic type (via deduction)



Variables

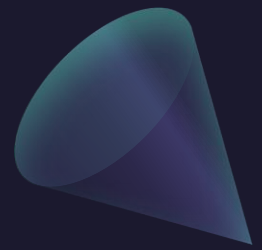
Part 2

Initialisation

WHAT ARE VARIABLES AND WHAT IS INITIALISATION

- A variable is an object or entity that has a single type and a single value.
- Variables store data for later use.
- Initialisation is the process of giving a variable a value of the variables type
- In C++, there are many ways to initialise a variable depending on the context.

KINDS OF INITIALISATIONS



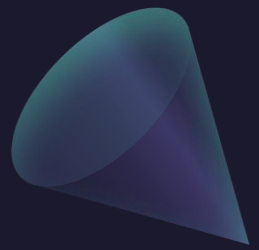
- Default
- Value
- Copy
- Direct
- Aggregate
- List

Qualifiers

SIGNED-NESS AND SIZE

- `signed` – Makes integral signed
- `unsigned` – Make integral unsigned
- `short` - Integral with at least 16-bits (2-bytes)
- `long` - Integral with at least 32-bits (4-bytes)
- `long long` - Integral with at least 64-bits (8-bytes)
- `unsigned` can be used in combination with the size qualifiers increase the maximum possible value.

STORAGE AND MUTABILITY



- `static` - Declares static storage
- `inline` - In-lines a function call
- `const` - Data is immutable
- `constexpr` - Data may be evaluated at compile time
- `volatile` – Data is likely to change outside the compilers insight.

Automatic Type Deduction

- C++ allows for the elision of type declaration through the use of type deduction.
- Type deduction takes the surrounding context of an expression and is able to infer what type a variable should be.
- Automatic types are introduced using the `auto` keyword.
- The type on the right-hand-side must be obvious to the compiler.
- E.g. `auto a = {1};` Here it is clear that `a` is an `int`.

Value Categories

LVALUES

- Found on the left-hand-side of the assignment operator (=).
- Indicates copy semantics when used in the right-hand side of =.

RVALUES

- Found on the right-hand-side of the assignment operator (=).
- Indicates a temporary value.
- Indicates move semantics.

Operators

Part 2

Basic Arithmetic

OPERATORS

- C++ has the typical operators you would expect to do basic arithmetic with integral and floating-point types.
- `+` - Addition and unary posigate
- `-` - Subtraction and unary negate
- `*` - Multiplication
- `/` - Division
- `%` - Modulus
- `++` - Increment (prefix and postfix)
- `--` - Decrement (prefix and postfix)

ABOUT DIVISION AND MODULO

- Division of two integral types will perform integer division, where the remained will be discarded.
- You must cast an integral to a floating point type.
- Modulo does not work on floating point types.

Basic Arithmetic Example 1

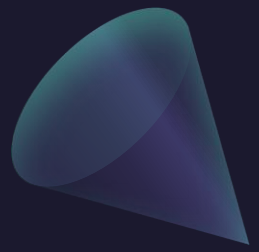
```cpp
#include <iostream>

auto main () -> int
{
    auto a{10};
    auto b{3};

    std::cout << "a + b = " << a + b << std::endl;  ///< a + b = 13
    std::cout << "a - b = " << a - b << std::endl;  ///< a - b = 7
    std::cout << "a * b = " << a * b << std::endl;  ///< a * b = 30
    std::cout << "a / b = " << a / b << std::endl;  ///< a / b = 3??
    std::cout << "a % b = " << a % b << std::endl;  ///< a % b = 1

    return 0;
}
```

Basic Arithmetic Example 2

```cpp
#include <iostream>

auto main () -> int
{
    auto a{10};
    auto b{3};
    auto c{3.};
    auto d{10.};

    std::cout << "a / c = " << a / c << std::endl;  ///< a / c = 3.33333
    std::cout << "d / b = " << d / b << std::endl;  ///< d / b = 3.33333

    return 0;
}
```

Basic Arithmetic Example 3

```cpp
#include <iostream>

auto main () -> int
{
    auto e{-5.43};
    auto f{0.71};

    std::cout << "e + f = " << e + f << std::endl;          ///< e + f = -4.72
    std::cout << "-e + f = " << -e + f << std::endl;        ///< -e + f = 6.14
    std::cout << "e - -f = " << e - f << std::endl;         ///< e - f = -6.14
    std::cout << "e - -f = " << e - -f << std::endl;        ///< e - -f = -4.72

    return 0;
}
```



Basic Arithmetic Example 4

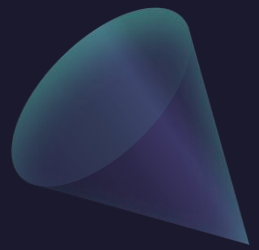
```cpp
#include <iostream>

auto main () -> int
{
    auto g{1};
    auto h{5};

    std::cout << "g++ = " << g++ << std::endl;      ///< g++ = 1
    std::cout << "g = " << g << std::endl;          ///< g = 2
    std::cout << "++g = " << ++g << std::endl;      ///< ++g = 3
    std::cout << "g = " << g << std::endl;          ///< g = 3

    std::cout << "h-- = " << h-- << std::endl;      ///< h-- = 5
    std::cout << "h = " << h << std::endl;          ///< h = 4
    std::cout << "--h = " << --h << std::endl;      ///< --h = 3
    std::cout << "h = " << h << std::endl;          ///< h = 3

    return 0;
}
```

Casting

- Casting allows for conversion of the type from and expression to a new type
- `const_cast<T>(/* expr */);` - Changes cv-qualifications
- `static_cast<T>(/* expr */);` - Converts type
- `reinterpret_cast<T>(/* expr */);` - Reinterprets the underlying bits
- `dynamic_cast<T>(/* expr */);` - Allows fir casting up, down and across the class hierarchies
- `static_cast<T>()` is the one we will use most.

Casting Example

```cpp
#include <iostream>

auto main () -> int
{
    auto a{10};
    auto b{3};

    /// Explicitly cast `b` to a `double`
    std::cout << "a / b = " << a / static_cast<double>(b) << std::endl;  ///< a / b = 3.33333

    return 0;
}
```



Bitwise Arithmetic

OPERATORS

- Bitwise operators allow for the manipulation of the underlying bits of a value in memory.
- & - And
- | - Or
- ^ - Xor
- << - Left Shift
- >> - Right Shift

FLOATING POINTS

- Bitwise operators only work for integral types.
- They do not work for floating point types.

Bitwise Arithmetic Example 1

```cpp
#include <bitset>
#include <iostream>

auto main () -> int
{
    auto i{5};
    auto j{4};

    std::cout << "i & j = " << (i & j) << std::endl;                    ///< i & j = 4
    std::cout << "  " << std::bitset<8>{i} << std::endl;
    std::cout << "& " << std::bitset<8>{j} << std::endl;
    std::cout << "----------" << std:: endl;
    std::cout << "  " << std::bitset<8>{i & j} << std::endl;             ///< i & j =  00000100
    return 0;
}
```

Bitwise Arithmetic Example 2

```cpp
#include <bitset>
#include <iostream>

auto main () -> int
{
    auto i{5};
    auto j{4};

    std::cout << "i | j = " << (i | j) << std::endl;                    ///< i | j = 4
    std::cout << "  " << std::bitset<8>{i} << std::endl;
    std::cout << "| " << std::bitset<8>{j} << std::endl;
    std::cout << "----------" << std:: endl;
    std::cout << "  " << std::bitset<8>{i | j} << std::endl;            ///< i | j =  00000101

    return 0;
}
```

Bitwise Arithmetic Example 3

```cpp
#include <bitset>
#include <iostream>

auto main () -> int
{
    auto i{5};
    auto j{4};

    std::cout << "i ^ j = " << (i ^ j) << std::endl;                    ///< i ^ j = 4
    std::cout << "  " << std::bitset<8>{i} << std::endl;
    std::cout << "^ " << std::bitset<8>{j} << std::endl;
    std::cout << "----------" << std:: endl;
    std::cout << "  " << std::bitset<8>{i ^ j} << std::endl;              ///< i ^ j =  00000001

    return 0;
}
```

Bitwise Arithmetic Example 4

```cpp
#include <bitset>
#include <iostream>

auto main () -> int
{
    auto i{5};
    auto j{4};

    std::cout << "i << j = " << (i << j) << std::endl;                    ///< i << j = 4
    std::cout << "    " << std::bitset<8>{i} << std::endl;
    std::cout << "<< " << std::bitset<8>{j} << std::endl;
    std::cout << "-----------" << std:: endl;
    std::cout << "    " << std::bitset<8>{i << j} << std::endl;           ///< i << j =  01010000

    return 0;
}
```

Bitwise Arithmetic Example 5

```cpp
#include <bitset>
#include <iostream>

auto main () -> int
{
    auto i{5};
    auto j{4};

    std::cout << "i >> j = " << (i >> j) << std::endl;                    ///< i >> j = 4
    std::cout << "    " << std::bitset<8>{i} << std::endl;
    std::cout << ">> " << std::bitset<8>{j} << std::endl;
    std::cout << "-----------" << std:: endl;
    std::cout << "    " << std::bitset<8>{i >> j} << std::endl;          ///< i >> j =  00000000

    return 0;
}
```

Arithmetic Assignment

- In C++, there are also assignment variants of all the arithmetic operators that perform the binary operation and then assign the result to the left point (argument).

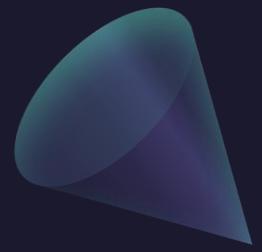
- `+=` - Add assign
- `-=` - Sub assign
- `*=` - Multiply assign
- `/=` - Divide assign
- `%=` - Modulo assign
- `&=` - And assign
- `|=` - Or assign
- `^=` - Xor assign
- `<<=` - Left Shift assign
- `>>=` - Right shift assign

Arithmetic Assignment Example

```cpp
#include <iostream>

auto main () -> int
{
    auto k{5};
    auto l{2};

    k += l;
    std::cout << "k += l -> k = " << k << std::endl;    ///< k = 7

    k *= l;
    std::cout << "k *= l -> k = " << k << std::endl;    ///< k = 14

    l |= k;
    std::cout << "l |= k -> l = " << l << std::endl;    ///< l = 14

    k <<= l;
    std::cout << "k <<= l -> k = " << k << std::endl;   ///< k = 229376

    l ^= k;
    std::cout << "l ^= k -> l = " << l << std::endl;    ///< l = 229390

    k &= l;
    std::cout << "k &= l -> k = " << k << std::endl;    ///< k = 229376

    l -= k;
    std::cout << "l -= k -> l = " << l << std::endl;    ///< l = 14

    return 0;
}
```

Size operator

- You can obtain the size of a type; in bytes using the `sizeof()` operator.
- This returns a `std::size_t` type.



Sizeof Operator Example

```cpp
#include <iostream>

auto main () -> int
{
    auto a {10};
    auto b {3.5};
    auto c {'c'};

    std::cout << "sizeof (a) = " << sizeof (a) << std::endl;  ///< sizeof (a) = 4
    std::cout << "sizeof (b) = " << sizeof (b) << std::endl;  ///< sizeof (b) = 8
    std::cout << "sizeof (c) = " << sizeof (c) << std::endl;  ///< sizeof (c) = 1

    return 0;
}
```

IO

Part 2

Character Streams

WHAT ARE STREAMS?

- Streams are a sequential buffer of elements.
- Streams connect your program to various IO devices.
- Streams are used to take input, write to files and control any form of buffered output.
- Streams can have manipulators composed within the stream.

STREAM OBJECTS

- `std::cout` - Mounted to C standard output.
- `std::cin` - Mounted to C standard input.
- `std::cerr` - Mounted to C standard error.
- `std::clog` - Mounted to C standard error (doesn't depend on `stdout`).

Character Streams Example 1

```cpp
#include <iostream>

auto main () -> int
{
    auto a {0};
    auto b {0};

    std::cout << "Enter two numbers: ";
    std::cin >> a >> b;

    std::cout << "a = " << a << std::endl;
    std::cout << "b = " << b << std::endl;
    std::cout << "a + b = " << (a + b) << std::endl;

    return 0;
}
```

Character Streams Example 2

```cpp
#include <iomanip>
#include <iostream>

auto main () -> int
{
    auto a {255};
    auto b {0.01};

    std::cout << "a: oct = " << std::oct << a << std::endl;  ///< 377
    std::cout << "a: hex = " << std::hex << a << std::endl;  ///< ff
    std::cout << "a: dec = " << std::dec << a << std::endl;  ///< 255

    std::cout << std::fixed << b << std::endl;           ///< 0.010000
    std::cout << std::scientific << b << std::endl;      ///< 1.000000e-02
    std::cout << std::hexfloat << b << std::endl;        ///< 0x1.47ae147ae147bp-7
    std::cout << std::defaultfloat << b << std::endl;    ///< 0.01

    return 0;
}
```

Equality, Ordering and Logical Operators

Part 2

Equality and Ordering

EQUALITY

- C++ has a strict sense of equality
- Equality and inequality are checked using binary operators
- `==` - Returns `true` if they are equal, otherwise `false`
- `!=` - Returns `false` if they are equal, otherwise `true`

ORDERING

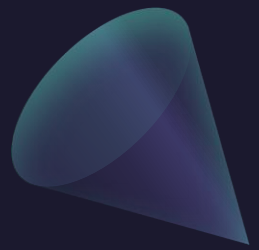
- Ordering is the notion of how objects relate to each other, e.g. order of numbers
- There are four ordering operators in C++
- Operators are read from left to right
- `<` - Less than
- `>` - Greater than
- `<=` - Less than or equal
- `>=` - Greater than or equal

Spaceships!

THREE-WAY COMPARISON OPERATOR

- There is another comparison operator in C++ called the three-way comparison (or spaceship) operator.
- `<=>` - Returns one of three ordering categories
- Each ordering category holds a variant state indicating the result of the comparison
- These states can one `-1`, `0` or `1` as an implicit value

ORDERING CATEGORIES



- The three ordering categories are
  - `std::strong_ordering`
  - `std::weak_ordering`
  - `std::partial_ordering`
- Each category having there own set of preconditions about the properties of the types that were compared

Logical Operators

EQUALITY

- These operators compare the values of Boolean expressions
- The binary logical operators have short circuiting properties allowing faster execution
- `!` – Logical Not, inverts Boolean value (unary)
- `&&` - Logical And
- `||` - Logical Or

LOGICAL XOR

- You may wonder why there is no logical Xor in C++ (`^^`). This is for two reasons.
- Short circuiting can not occur for Xor based operations, both points must be evaluated
- The truth table of a logical Xor like operation can occur using the expression `!(a) != !(b)`

Equality, Ordering and Logical Operators Example 1

```cpp
#include <iomanip>
#include <iostream>

auto main () -> int
{
    auto a {1};
    auto b {2};

    std::cout << std::boolalpha;
    std::cout << "a == b => " << (a == b) << std::endl;  ///< false
    std::cout << "a != b => " << (a != b) << std::endl;  ///< true

    std::cout << "a == a => " << (a == a) << std::endl;  ///< true
    std::cout << "a != a => " << (a != a) << std::endl;  ///< false
    std::cout << std::noboolalpha;
    return 0;
}
```

Equality, Ordering and Logical Operators Example 2

```cpp
#include <iomanip>
#include <iostream>

auto main () -> int
{
    auto a {1};
    auto b {2};

    std::cout << std::boolalpha;
    std::cout << "a < b => " << (a < b) << std::endl;       ///< true
    std::cout << "a > b => " << (a > b) << std::endl;       ///< false

    std::cout << "a <= a => " << (a <= a) << std::endl;     ///< true
    std::cout << "a >= a => " << (a >= a) << std::endl;     ///< true

    std::cout << "a <= b => " << (a <= b) << std::endl;     ///< true
    std::cout << "a >= b => " << (a >= b) << std::endl;     ///< false
    std::cout << std::noboolalpha;
    return 0;
}
```

Equality, Ordering and Logical Operators Example 3

```cpp
#include <compare>
#include <iomanip>
#include <iostream>

auto main () -> int
{
    auto a {1};
    auto b {2};

    auto aa = a <=> a;
    auto ab = a <=> b;

    std::cout << std::boolalpha;
    std::cout << "((a <=> a) < 0) => " << ((a <=> a) < 0) << std::endl;      ///< false
    std::cout << "((a <=> a) == 0) => " << ((a <=> a) == 0) << std::endl;    ///< true
    std::cout << "((a <=> a) > 0) => " << ((a <=> a) > 0) << std::endl;      ///< false

    std::cout << "((a <=> b) < 0) => " << ((a <=> b) < 0) << std::endl;      ///< true
    std::cout << "((a <=> b) == 0) => " << ((a <=> b) == 0) << std::endl;    ///< false
    std::cout << "((a <=> b) > 0) => " << ((a <=> b) > 0) << std::endl;      ///< false

    return 0;
}
```

Equality, Ordering and Logical Operators Example 4

```cpp
#include <iomanip>
#include <iostream>

auto main () -> int
{
    auto a {1};
    auto b {2};
    auto c {3};

    std::cout << std::boolalpha;

    /// if `a` is less than `b` and if `a` is less than `c`
    std::cout << "((a < b) && (a < c)) => " << ((a < b) && (a < c)) << std::endl;       ///< true

    /// if `c` is greater than `b` or if `a` is greater than `c`
    std::cout << "((c > b) || (a > c)) => " << ((c > b) || (a > c)) << std::endl;       ///< true

    /// if `a` is not greater than `b` or if `a` is equal to `c`
    std::cout << "(!(a > b) || (a == c)) => " << (!(a > b) || (a == c)) << std::endl;   ///< true
    /// if `a` is not greater than `b` is not equal to if `a` is not greater than `c`
    /// if `a` is not greater than `b` xor if `a` is not greater than `c`
    std::cout << "(!(a > b) != !(a < c)) => " << (!(a > b) != !(a > c)) << std::endl;   ///< false

    std::cout << std::noboolalpha;

    return 0;
}
```

Conditionals

Part 2

Scope

WHAT IS SCOPE?

- Scope is a way to separate different logical sections of code.
- Scope blocks (or code blocks) are denoted by a pair of braces { }.

EXAMPLE

```
#include <iostream>

auto main () -> int
{
    auto a {4};

    {
        auto b {6};
        std::cout << a << std::endl;
        std::cout << b << std::endl;
    }

    std::cout << a << std::endl;
    std::cout << b << std::endl;  ///< Will fail here, comment out
to run

    return 0;
}
```

Conditional Expressions

IF-EXPRESSIONS AND ELSE-CLAUSE

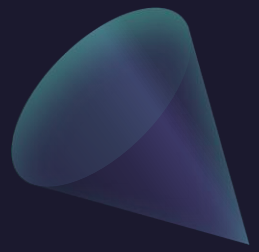
- if-expressions allows for section code to be run conditionally
- Can be used in combination with an else-clause to create a two variant branch in your program
- Encodes ‘if something is true, do this, else do this’.

ELSE-IF-EXPRESSIONS

- You can combine else-clause with an if-expression to create an else-if-expression.
- Allows for multiple conditions to be check in series.

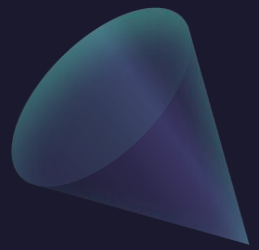
Conditional Expressions Example 1

```cpp
#include <iostream>

auto main () -> int
{
    auto a {1};
    auto b {2};

    if (a < b)
    {
        std::cout << "a is less then b" << std::endl;
    }

    if (a == b)
        std::cout << "a is equal to b" << std::endl;

    if (a > b)
        std::cout << "a is greater then b" << std::endl;

    return 0;
}
```



Conditional Expressions Example 2

```cpp
#include <iostream>

auto main () -> int
{
    auto a {1};
    auto b {2};

    if (a == b)
        std::cout << "a is equal to b" << std::endl;
    else
        std::cout << "a is not equal to b" << std::endl;

    return 0;
}
```



Conditional Expressions Example 3

```cpp
#include <iostream>

auto main () -> int
{
    auto a {1};
    auto b {2};

    if (a < b)
        std::cout << "a is less then b" << std::endl;
    else if (a == b)
        std::cout << "a is equal to b" << std::endl;
    else if (a > b)
        std::cout << "a is greater then b" << std::endl;
    else
        std::cout << "a is unordered to b" << std::endl;

    return 0;
}
```

Loops

Part 2

While Loops

WHILE LOOP

- Repeats a give section of code as long as a condition is met
- Condition is checked at the start of every loop
- Can be escaped with a break-term or return-expression.

DO-WHILE LOOP

- Repeats a give section of code as long as a condition is met
- Condition is checked at the end of every loop
- Loop runs at least once
- Can be escaped with a break-term or return-expression.

For Loops

FOR LOOP

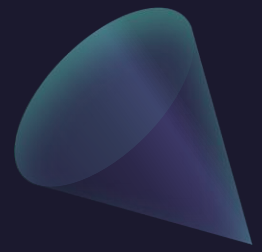
- Similar to a while-loop but encodes the initialiser, conditional and state change in a single expression
- Used to iteratively move through a (usually) numeric range

RANGE-FOR LOOP

- Yields a new value from a range of elements
- Continuous through range until it is exhuasted

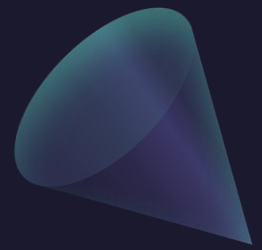
Loops Example 1

```cpp
#include <iostream>

auto main () -> int
{
    auto a {10};

    while (a > 0)
    {
        std::cout << "a = " << a << std::endl;
        --a;
    }

    return 0;
}
```



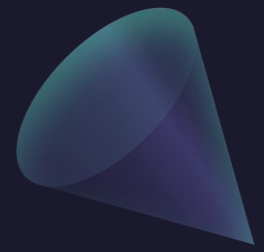
Loops Example 2

```cpp
#include <iostream>

auto main () -> int
{
    auto a {0};

    do
    {
        std::cout << "a = " << a << std::endl;
        --a;
    } while (a > 0);

    return 0;
}
```



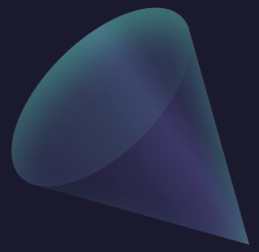
Loops Example 3

```cpp
#include <iostream>

auto main () -> int
{
    for (auto i {0}; i < 10; ++i)
        std::cout << "i = " << i << std::endl;

    return 0;
}
```



Loops Example 4

```cpp
#include <iostream>

auto main () -> int
{
    std::cout << "[ ";
    for (auto i : {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})
        std::cout << i << ", ";

    std::cout << "]" << std::endl;

    for (auto s : {"It's over Anakin!", "I have the high ground!"})
        std::cout << s << std::endl;

    return 0;
}
```



Functions

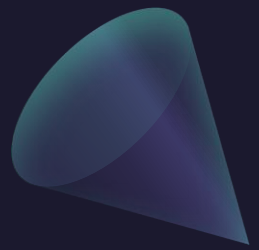
Part 2

Functions

- Functions allow for the encapsulation of code
- Functions are the most basic more of abstraction in all of computer science
- They allow for repeated use of the same section of code
- Functions allow for data to be manipulated efficiently and concisely
- Reduces code complexity by breaking down a system into various components
- Functions can have side effects that is not evident from just its signature
- Functions can return nothing using the `void` keyword.

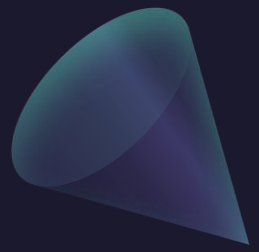
Functions Example 1

```cpp
#include <iostream>

int add(int x, int y)
{ return x + y; }

auto main () -> int
{
    auto a {4};
    auto b {7};
    auto c {-3};

    int d = add(5, 7);

    std::cout << add(a, b) << std::endl;
    std::cout << add(a, c) << std::endl;
    std::cout << add(a, d) << std::endl;
    std::cout << add(b, c) << std::endl;
    std::cout << add(b, d) << std::endl;
    std::cout << add(c, d) << std::endl;

    return 0;
}
```



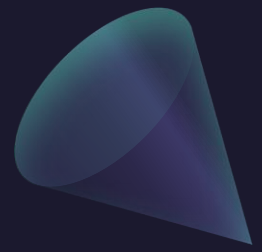
Functions Example 2

```cpp
#include <iostream>

int sum(int s, int f)
{
    auto acc {0};
    for (auto i {s}; i < f; ++i)
        acc += i;
    return acc;
}

auto main () -> int
{

    std::cout << sum(0, 5) << std::endl;
    std::cout << sum(-3, 8) << std::endl;
    std::cout << sum(-11, -5) << std::endl;
    std::cout << sum(4, 19) << std::endl;

    return 0;
}
```



Functions Example 3

```
#include <iostream>

void println(auto s)
{ std::cout << s << std::endl; }

auto main () -> int
{
    println("Hello World!");

    return 0;
}
```



Discussion

- Any questions?
- Need help?
- Open discussion.
- Concerns?

Next Part

Pointers

Slices

References

Dynamic Memory

The Standard Library

Summary

This week you learnt about C++'s type system, what variables are and how to perform actions using operators. We also looked at ordering, equality conditional expressions and functions. We also looked at IO in C++ and looping.

Thank You

Tyler Swann

https://github.com/MonashDeepNeuron/HPP