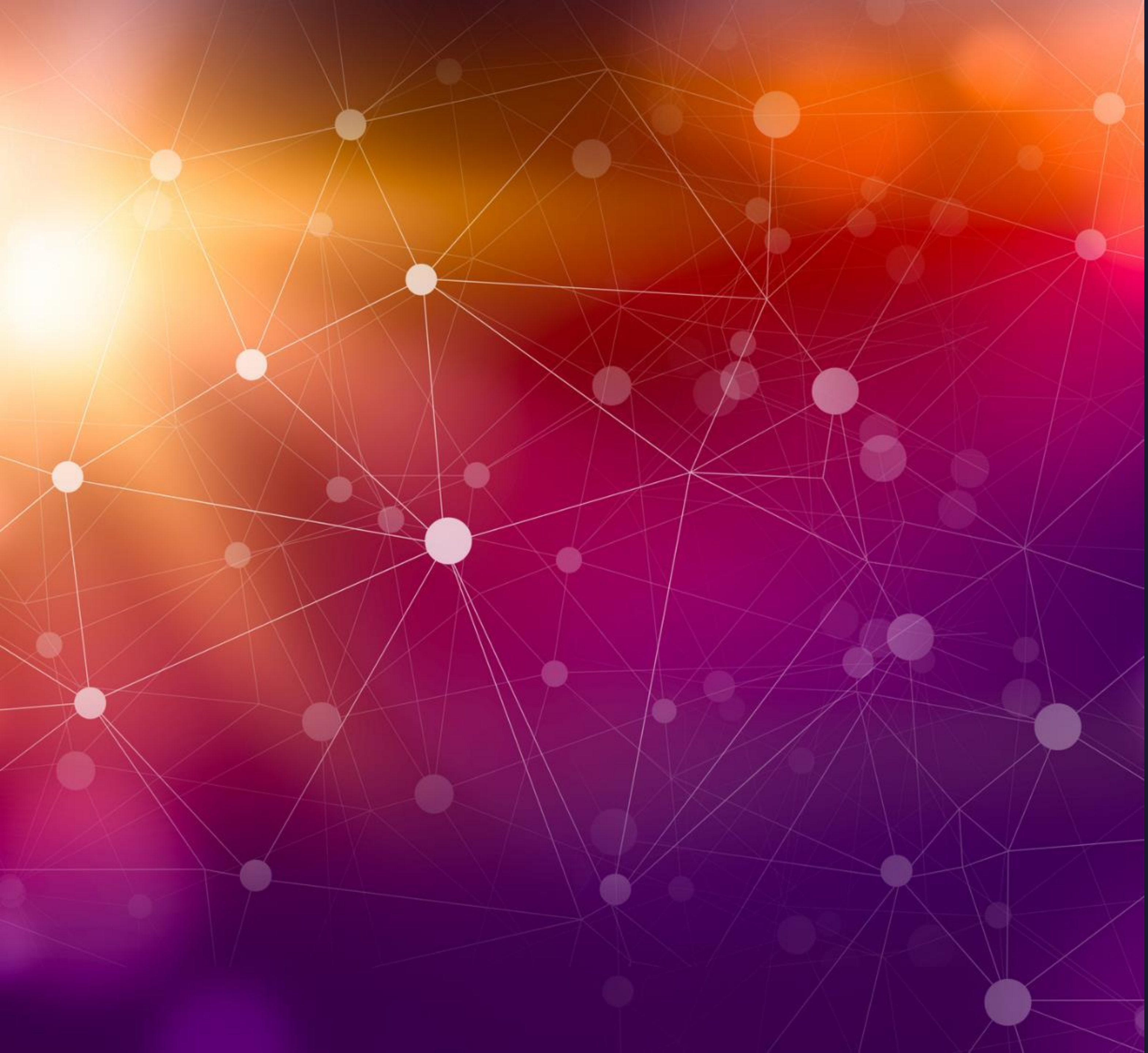


Part 4 – Advanced Functions

Tyler Swann

Agenda

- Advanced Functions
- Namespaces
- Enumerations
- Unions
- Structures
- Discussion
- Assignment One Showcase

Advanced Functions

Part 4

Function Specification

- Marking a function as *noexcept* can improve the execution path prediction (branch prediction) of functions by marking them as to not throw an exception.
- Attributes allow for specify certain conditions or facts about a function including whether it returns a value, how parameters are used and more.
- The return type of a function can be declared at the end of a function signature (as opposed to in front) using *auto* as a placeholder and using an arrow (->) followed by the type. This helps to declare a return type is dependent on the types of the function's parameters.

```cpp
[[nodiscard]]
auto f(int n) noexcept
    -> int
{ return n; }
```

Overloading

FUNCTION OVERLOADING

- The type signature of functions in C++ are a combination of the functions name, return type and the types of its parameter due to name mangling in C++.
- Because of name mangling, functions with the same name but different parameter types can exist.

OPERATOR OVERLOADING

- Operators can also be overloaded to support custom types and operations.
- This is how `std::cout` supports stream building using `<<`.

Utility Function

PERFECT FORWARDING

- Because C++ has a strict notion of value categories of types, certain operations can lead to underside effects, e.g. passing parameters from a wrapper type to an inner function.
- C++ has a function, `std::forward<T>()`, for perfectly forwarding objects while maintaining the value category of the object

TYPE AND VALUE DECLARATORS

- The type of an object can be declared using `decltype()`.
- You can construct a rvalue default constructed object that doesn't support rvalue default construction with `std::declval<T>()`. This is useful for type introspection of member types and methods.

Functional Programming

FUNCTION TYPES

- Sometimes it is useful to pass functions as objects.
- This is most useful for passing implementation algorithms or functors to other proxy functions.
- Functions can be represented using the type `std::function<R(Args...)>`.

LAMBDAS AND CLOSURES

- Closures can capture the surrounding contextual environment and move them between other environments.
- Lambdas create closures by capturing certain objects into their local environment while also acting as function types.

Functional Programming

PARTIAL APPLICATION

- Sometimes, only certain information is known that a function might need in order to be invoked.
- Using `std::bind()`, we can partially apply parameters to a function and create a new function that will invoke the function with the remaining necessary parameters.
- `std::bind()` supports positional argument application, allow you to specify where parameters applied later with be called in the original function.
- `std::bind_front()` and `std::bind_back()` can also be used for more specific use cases.

Overloading Example 1

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

using namespace std::literals;

/// Comment out to see old behaviour
auto operator+ (std::string x, std::string y) -> int
{ return std::stoi(x) + std::stoi(y); }

auto main() -> int
{
    std::cout << "6"s + "5"s << std::endl;

    return 0;
}
```

Overloading Example 1

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

using namespace std::literals;

/// Comment out to see old behaviour
auto operator+ (std::string x, std::string y) -> int
{ return std::stoi(x) + std::stoi(y); }

auto main() -> int
{
    std::cout << "6"s + "5"s << std::endl;

    return 0;
}
```

Overloading Example 1

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

using namespace std::literals;

/// Comment out to see old behaviour
auto operator+ (std::string x, std::string y) -> int
{ return std::stoi(x) + std::stoi(y); }

auto main() -> int
{
    std::cout << "6"s + "5"s << std::endl;

    return 0;
}
```

Function Utilities Example 1

```cpp
#include <type_traits>

auto add(int x, float y) -> decltype(x + y)
{ return x + y; }

auto main() -> int
{
    static_assert(std::is_same_v<decltype(add(9, 0.345)), float>, "Result is not a float");
    static_assert(std::is_same_v<decltype(add(9, 0.345)), int>, "Result is not a int");

    return 0;
}
```

Function Utilities Example 2

```cpp
#include <type_traits>

auto main() -> int
{
    static_assert(std::is_same_v<decltype(std::declval<int>()), int&&>, "Result is rvalue");
    static_assert(std::is_same_v<decltype(std::declval<int>()), int>, "Result is not rvalue");

    return 0;
}
```

Functional Programming Example 1

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

auto print_num(double i)
    -> void
{ std::cout << i << '\n'; }

auto add(int x, float y)
    -> double
{ return x + y; }

auto mult_print(int x, float y, std::function<void(double)> f)
    -> void
{ f(x * y); }

auto main() -> int
{
    std::function<void(double)> print_f {print_num};
    std::function<double(int, float)> add_f {add};

    print_f(add_f(4, 6.6));
    mult_print(3, 0.056, print_f);

    return 0;
}
```

Functional Programming Example 2

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

auto fmult(int x, float y, std::function<void(double)> f)
    -> void
{ f(x * y); }

auto main() -> int
{
    /// Use `std::function<R(Args...)>` for lambda type
    std::function<double(int, float)> add_f = [](int x, float y) -> double { return x + y; };

    /// Lambda declared with `auto`
    auto print_f = [](double i){ std::cout << i << '\n'; };

    /// Lambda capture `print_f` by value
    auto print_mult = [=](int x, float y){ return fmult(x, y, print_f); };

    int a {7};
    int b {5};

    /// Capture `print_mult` and `a` by value and `b` by reference
    /// Elide names of `mult_print` and `a` with `=`
    auto print_7mult5 = [=, &b](){ return print_mult(a, b); };

    /// Invoke lambdas like functions
    print_f(add_f(4, 6.6));
    fmult(7, 8.9, [](double i){ std::cout << i << '\n'; });     ///< Use lambda as anonymous function,
    print_mult(7, 8.9);                                          ///< or use captured version


    print_7mult5();     ///< 35
    b = 9;              ///< Modify `b`
    print_7mult5();     ///< 63

    return 0;
}
```

Functional Programming Example 3

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

auto fn(int n1, int n2, int n3, const int& n4, int n5)
{
    std::cout << n1 << ' '
              << n2 << ' '
              << n3 << ' '
              << n4 << ' '
              << n5 << '\n';
}

/// Import placeholders
using namespace std::placeholders;

auto main() -> int
{
    auto f1 = std::bind(fn, 1, _1, 4, _2, 6);
    auto f2 = std::bind(fn, _1, _1, _1, _1, _1);

    auto a {47676};
    auto f3 = std::bind(fn, _4, _3, _2, std::cref(a), _1);

    f3(4, 3, 2, 1);     ///< 1 2 3 47676 4
    f1(4, a);           ///< 1 4 4 47676 6
    a = 777;
    f3(11, 10, 9, 8);   ///< 8 9 10 777 11
    f1(3, a);           ///< 1 3 4 777 6

    f2(6);              ///< 6 6 6 6 6

    return 0;
}
```

Functional Programming Example 4

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

auto fn(int n1, int n2, int n3, const int& n4, int n5)
{
    std::cout << n1 << ' '
              << n2 << ' '
              << n3 << ' '
              << n4 << ' '
              << n5 << '\n';
}

auto main() -> int
{
    auto f = std::bind_front(fn, 1, 2, 3);
    f(4, 5);  ///< 1 2 3 4 5

    return 0;
}
```

Namespaces

Part 4

Namespaces

- Namespaces allow for the logical separation of symbols.
- Namespaces are access using the scope resolution operator `::`.
- The global namespace is access using `::<symbol>`.
- Two namespaces with the same name will logically merge allow for building up a namespace across source files.
- Namespaces can also be nested.

Namespaces Example 1

```cpp
#include <iostream>

namespace A
{
    auto f(int n)
        -> void
    { std::cout << n << '\n'; }
}

auto f(int n)
    -> void
{ std::cout << "3 + n:(" << n << ") = " << 3 + n << '\n'; }

auto main() -> int
{
    // using namespace A;  ///< Error: overload is ambiguous (redefinition)
    f(8);
    A::f(8);

    return 0;
}
```

Namespaces Example 2

```cpp
#include <iostream>

namespace A
{
    auto f(int n)
        -> void
    { std::cout << n << '\n'; }
}

namespace A
{
    auto g(int n)
        -> void
    { std::cout << "3 + n:(" << n << ") = " << 3 + n << '\n'; }
}

auto main() -> int
{
    A::f(8);
    A::g(8);

    return 0;
}
```

Namespaces Example 3

```cpp
#include <iostream>

namespace A
{
    auto f(int n)
        -> void
    { std::cout << n << '\n'; }

    namespace B
    {
        auto f(int n)
            -> void
        { std::cout << "3 + n:(" << n << ") = " << 3 + n << '\n'; }
    }
}

auto main() -> int
{
    A::f(8);
    A::B::f(8);

    return 0;
}
```

Enumerations

Part 4

Enumerations

- Enumerations (enums) are a distinct types whose value is one of a restricted range of named integral constants called enumerators.
- Enums allow for specify a type that may have a value of one of many possible named values.
- The underlying type of the enumerators is some integral type, usually `int`. The underlying type can be specified.
- Enumerators can have a specific value or will increment from 0 (or from the last specified value).
- Enumerators can be restricted to be only access using `::` making them more type safe.

Enumerations Example 1

```cpp
#include <iostream>

enum Colour { Red, Green, Blue};

auto print_colour_name(Colour c)
    -> void
{
    switch (c)
    {
        case Red:
            std::cout << "Red\n";
            break;
        case Green:
            std::cout << "Green\n";
            break;
        case Blue:
            std::cout << "Blue\n";
            break;
        default:
            std::cout << "Not a colour\n";
            break;
    }
}

auto main() -> int
{
    Colour c1 {Red};             ///< Unscoped Initialisation
    Colour c2 {Colour::Green};   ///< Scoped Initialisation
    auto c3 {Colour::Blue};      ///< `auto` type deduction
    auto c4 {4};                 ///< Non `Colour`

    print_colour_name(c1);
    print_colour_name(c2);
    print_colour_name(c3);
    print_colour_name(static_cast<Colour>(c4));

    return 0;
}
```

Enumerations Example 2

```cpp
#include <iostream>

/// Enum `Colour` whose underlying type is `short`
enum Colour : short
{ Red, Green = 57, Blue};

auto print_colour_name(Colour c)
    -> void
{
    switch (c)
    {
        case Red:
            std::cout << "Red = ";
            break;
        case Green:
            std::cout << "Green = ";
            break;
        case Blue:
            std::cout << "Blue = ";
            break;
        default:
            std::cout << "Not a colour\n";
            return;
    }

    std::cout << static_cast<short>(c) << std::endl;
}

auto main() -> int
{
    Colour c1 {Red};             ///< Unscoped Initialisation
    Colour c2 {Colour::Green};  ///< Scoped Initialisation
    auto c3 {Colour::Blue};     ///< `auto` type deduction
    auto c4 {4};                ///< Non `Colour`

    print_colour_name(c1);
    print_colour_name(c2);
    print_colour_name(c3);
    print_colour_name(static_cast<Colour>(c4));

    return 0;
}
```

Enumerations Example 3

```cpp
#include <iostream>

/// Enum class `Colour` whose underlying type is `short`
enum class Colour : short
{ Red, Green = 57, Blue};

auto print_colour_name(Colour c)
    -> void
{
    switch (c)
    {
        case Colour::Red:
            std::cout << "Red = ";
            break;
        case Colour::Green:
            std::cout << "Green = ";
            break;
        case Colour::Blue:
            std::cout << "Blue = ";
            break;
        default:
            std::cout << "Not a colour\n";
            return;
    }

    std::cout << static_cast<short>(c) << std::endl;
}

auto main() -> int
{
    // Colour c1 {Red};         ///< Unscoped Initialisation (error for `enum class`)
    Colour c2 {Colour::Green};  ///< Scoped Initialisation
    auto c3 {Colour::Blue};     ///< `auto` type deduction
    auto c4 {4};                ///< Non `Colour`

    // print_colour_name(c1);
    print_colour_name(c2);
    print_colour_name(c3);
    print_colour_name(static_cast<Colour>(c4));

    return 0;
}
```

Unions

Part 4

Unions

- Unions are a special kind of type known as an algebraic data type. This means the type of a union object can vary between a small list of possible types.
- Unions allow for a single type represent many possible types that can change throughout the lifetime of a union object. The members of a union occupy the same memory space, thus the size of a union is the size of the largest possible member.
- Constructing a union object will always need to construct the first variant.
- Accessing the non-activate member is UB.

Unions Example 1

```cpp
#include <iostream>

union Sym
{
    int num;
    float float32;
    const char* str;
};

auto main() -> int
{
    Sym sym {8};
    std::cout << sym.num << std::endl;

    sym.float32 = 5.6f;
    std::cout << sym.float32 << std::endl;

    sym.str = "Hello";
    std::cout << sym.str << std::endl;

    return 0;
}
```

Unions Example 2

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

union S
{
    std::string str;
    std::array<int, 5> arr;
    ~S() {}                 ///< Variant `str` has non-trivial destructor
};

int main()
{
    S s = {"Hello, world"};

    std::cout << "s.str = " << s.str << '\n';
    s.str.~basic_string();                      ///< Explicity destroy string

    s.arr = std::array<int, 5>{1, 2, 3, 4, 5};  ///< Explicity create array
    s.arr[1] = 5675;                            ///< Assign 2nd element to 3

    for (auto& v : s.arr)
        std::cout << v << ' ';
    std::cout << std::endl;

}
```

Option & Variant

- `std::optional<T>` represents an algebraic data type that either holds a value or holds nothing.
- `std::variant<Ts...>` is a type safe version of union, implemented as a tagged union.
- These types allow for introspection of the state of the object such as whether it holds a valid object or accessing the object itself.

Option Example

```cpp
#include <cmath>
#include <limits>
#include <iostream>
#include <optional>
#include <string>

auto divide(int x, int y)
    -> std::optional<float>
{
    if (y == 0)
        return std::nullopt;

    return std::optional<float>{x / static_cast<float>(y)};
}

auto main() -> int
{

    auto opt1 = divide(4, 5);
    std::cout << opt1.value() << std::endl;

    /// Given `opt2` and `opt3` have the value `std::nullopt`
    /// the value passed to `.value_or()` is returned
    auto opt2 = divide(2, 0);
    std::cout << opt2.value_or(std::numeric_limits<float>::quiet_NaN()) << std::endl;

    auto opt3 = divide(4656, 0);
    std::cout << opt3.value_or(0.1f) << std::endl;

    return 0;
}
```

Variant Example

```cpp
#include <iostream>
#include <string>
#include <variant>
#include <vector>

/// Used to perform pattern matching
template<class... Ts> struct match : Ts... { using Ts::operator()...; };

using Sym = std::variant<int, float, std::string, long>;

auto main() -> int
{
    std::vector<Sym> syms = {8, "Hello", 6.8f, 4, "Bye", 857565L};
    for (auto& var : syms)
    {
        std::visit(match{
            [](int i){ std::cout << "Sym: <Integer> = " << i << std::endl; },
            [](float f){ std::cout << "Sym: <Float> = " << f << std::endl; },
            [](std::string s){ std::cout << "Sym: <String> = " << s << std::endl; },
            [](auto&& o){ std::cout << "Sym: <Other> = " << o << std::endl; }
        }, var);
    }

    return 0;
}
```

Structures

Part 4

Structures

- Structures are an integral part of C++. They allow for the creation of custom types.
- Structures are composed of member variables and member functions (methods).
- Structures are created using the `struct` keyword.
- Members are accessed using the member access operator `.` and the pointer member access operator `->`.

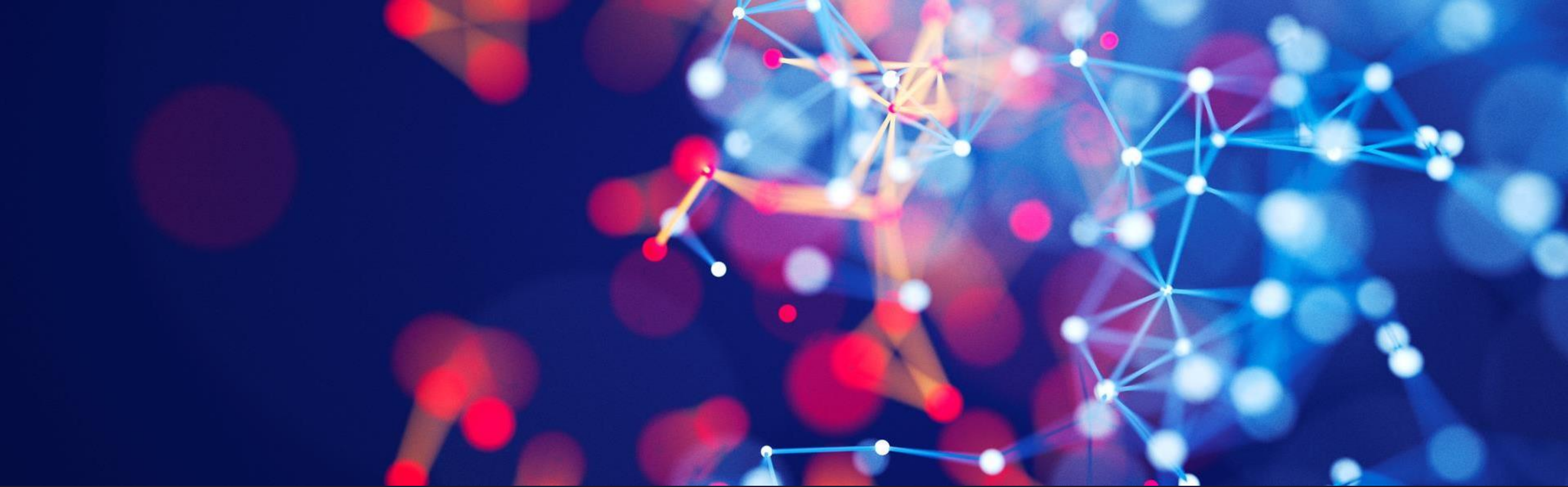
Structures Example

```cpp
#include <iostream>
#include <memory>

struct PairInt
{
    int first;
    int second;

    /// Adds members of two `PairInt`
    constexpr auto
    add(const PairInt& o)
        const noexcept
        -> PairInt
    { return PairInt{first + o.first, second + o.second}; }

    /// Overload `+~ cleaner `PairInt::add` call
    friend constexpr auto
    operator+ (const PairInt& x, const PairInt& y)
        noexcept
        -> PairInt
    { return x.add(y); }

    /// Overload `<<` for printing
    friend auto
    operator<< (std::ostream& os, const PairInt& v)
        -> std::ostream&
    {
        os << "(.first: " << v.first << ", .second: " << v.second << ")";
        return os;
    }
};

auto main() -> int
{
    auto a = PairInt{5, 7};
    auto b = PairInt{.first = 2, .second = 9};  ///< Named aggregate initialisation
    auto p = std::addressof(b);                 ///< Pointer to struct type

    std::cout << "a = " << a << std::endl;      ///< `PairInt` works with `std::cout`
    std::cout << "b = " << b << std::endl;

    std::cout << "a + b = " << a + b << std::endl;      ///< Call to overloaded `+`
    std::cout << "a + c = " << a + *p << std::endl;     ///< Pointer to structs works like regular pointers

    std::cout << "a.add(b) = " << a.add(b) << std::endl;    ///< Method access
    std::cout << "p->add(a) = " << p->add(a) << std::endl;  ///< Pointer member access

    return 0;
}
```

Discussion

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

Assignment 1



Summary

This week we delves deeper into the practical abilities of C++ from advanced function usage and functional programming idioms to logical code separation and modularization. You will also be introduced to a features that allow for the creation of custom types with the help of structures, enumerations and unions.

Next Week

- Classes
- Templets
- Concepts
- Standard Template Library

Thank You

Tyler Swann

https://github.com/MonashDeepNeuron/HPP