Topics for Today

- Revision
- Questions from last week
  - How to write C++ on my machine?
  - Why do we write `int a{};` ?
- C++ theory
  - Initialising variables
  - The standard library (std)
  - Preprocessor directives
  - Compilation (.cpp & .h)
  - Vectors
- This week's lab
  - Product of Array Except Self
  - Toy Vector

Pointers and References

As initialisers

int* x “Pointer”
Initialises a variable containing a memory address.
Contains:
0x00000000

int& x “Reference”
Initialises an *alias* for an existing variable. They both refer to the same memory address to read/write their data.
Contains: data

As operators

&x “Address of”
Get the address of this variable.
Returns:
0x00000000

*x “Dereference”
Gets the data stored in that memory address.
Used on pointers.
Returns: data

As function parameters

Function parameters use the initialiser rules from the previous table.

```
int myFunc(int& x){}
```

Gives you the original, editable variable in the function body.

```
int myFunc(const int& x){}
```

Same as above, but will not compile if you make any changes.

```
int myFunc(int x){}
```

Gives you a copy of x to use within the function body. Changes to x within the function do not affect the original variable.

```
int myFunc(int* x){}
```

Gives you a pointer, this is useful for objects that we store on the heap (talk about this later).

What did we learn last lab?

One thing from each corner of the room

What did we learn last lab?

```
// Printing:
std::cout << "Hello maties" << " I hope you've had a good week." << "\n";
```

```
// Receiving input:
// 1. Preallocate memory
std::string inputString{};
// 2. Stream extract into your allocated memory
std::cin >> inputString;
```

Q: Why do we put {} after our variables?

In C++, if we just make a variable like `int a;` then we are allocating space for it, but not doing anything to that space. So, anything that was stored there in RAM previously is still there. So C++ may not allow us to use it until it has been initialised.

Initialising with `int b{};` on the other hand, will allocate the memory and clear it out, making it safe to use.
It zero-initialises the variable for any type, ie works for string, float, pointers, etc.

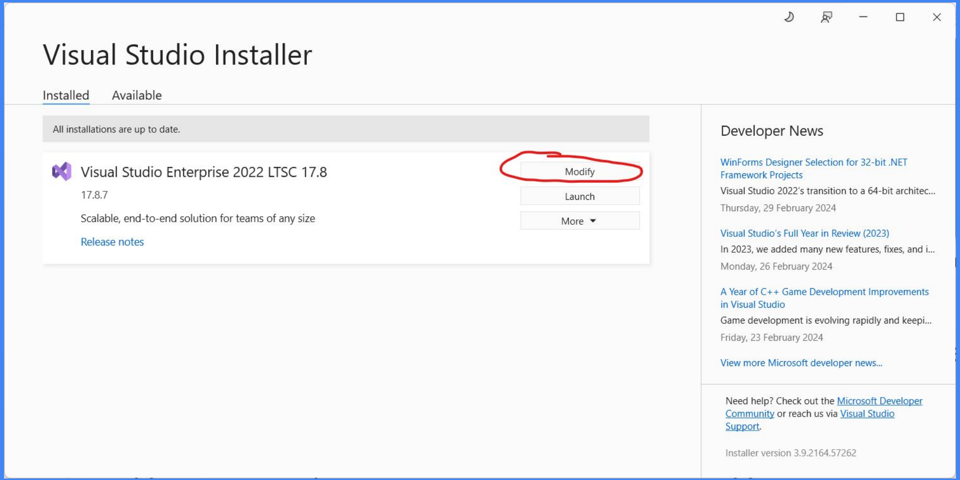
IDE/Text Editor and Compiler

There were some concerns last week about compiling C++ programs locally.

My advice is to install Visual Studio 2022, with the C++ workloads.
If you do this, you don't have to worry about setting everything up and get a good predictive IDE with a fantastic debugger out of the box.

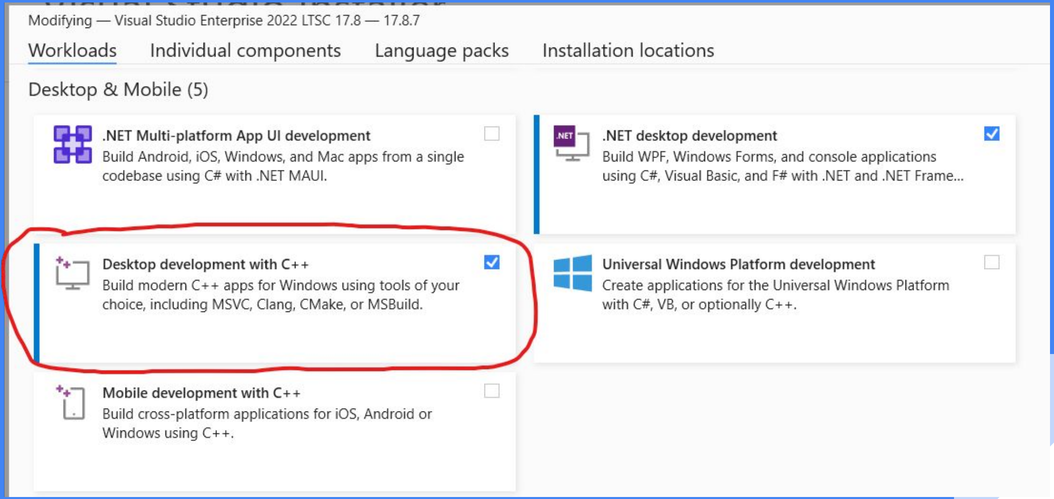
Text Editor:
An application that you can use to load and edit text files to write your code.
Use command line to do anything with it.
ie compile code, run code, etc

IDE:
A smart application that has more indepth features at the cost of being more expensive to run. Generally plug & play.
Handles anything command line you would need to do.



It's actually really easy.

You can also get a student license for Rider, but you still need to follow these steps.



Declaring and Initialising Variables

```
int num;
int num {};
int num {5};
int num = 5;
```

```
int  numbery{ 99 };
int* pointy = &numbery;
```

type name initialiser

```
std::vector<char> sillyString1;
std::vector<char> sillyString2 {};
std::vector<char> sillyString3 {'h','a','h','a'};
std::vector<char> sillyString4 (5, 'h');
std::vector<char> sillyString5 (sillyString3);
```

```
Size: 0
Size: 0
Size: 4 h, a, h, a,
Size: 5 h, h, h, h, h,
Size: 4 h, a, h, a,
```

type name initialiser

```
MyClass freddy{};
MyClass jonesy = MyClass("Jonesy");
MyClass* mrp = new MyClass("Mr. Pointy");
```

typename initialiser

```
glm::mat4 view = glm::mat4(1.0f);
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
```

type name initialiser

C++ has a different style as well

```
// C#/Java style
List<int> list = new List<int>();
```

```
// C++ options
MyClass henry("Henry");
MyClass greg("Gregor", 55);
MyClass* mitchell = new MyClass("Mitchell", 22);
MyClass* oscar{ new MyClass("Oscar", 73) };
```

type name initialiser

```
class MyClass {
public:
    int a;
    float b;
    std::string c;
    MyClass(int x, float y, std::string z) : a(x), b(y), c(z) {}
};

int main() {
    MyClass obj = { 42, 3.14f, "hello" };
    return 0;
}
```

C++ also has initialiser lists

The Standard Library

C++ is entirely backwards compatible with C. As such C has priority in naming conventions. To make C++ a more modern language, the standard library was introduced. Anything that is prefaced with `std::` is a standard library feature. *Primitive types comes from C and do not need `std::` ie `int`, `bool`, `float`*

Useful standard library features:

- `std::string` - Much nicer than old c-strings.
- `std::cout/cin` - useful I/O.
- `std::vector` - variable length arrays.
- `std::stack/queue`
- `std::list`
- `std::map`

When to expect to use `std`

C tends to be a more low level language. From C we get our basic data types, i.e. `int`, `char`, `float`, `int[]`, `int*`, `int&`, `sizeof()`, `*ptr`, etc

C++ is an extension on C that wants to have more high level function. For this case we can use the standard library, i.e. `std::string`, `std::vector`

Preprocessor Directives

Things beginning with

There are several stages to C/C++ compilation.

One of the first is preprocessing.

These make alterations to the source code for us.

For instance:

```
#define PI 3.1415926536

int main()
{
    std::sinf(2 * PI);
    // -> after preprocessing
    //    the compiler will read as:
    std::sinf(2 * 3.1415926536);
```

```
#if MACOS
    std::cout << "Running on a Mac \n";
#elif LINUX  #if MACOS
    std::cout << "Running on linux \n";
#else  #elif LINUX
    std::cout << "Running on windows \n";
#endif  #elif LINUX #else
```

#include

Note: we use
<library> and “localFile”

We have mostly seen `#include`
so far in this course.
What does it do?

It copies and pastes the contents
of a file into a file for us.

Example:

add.cpp

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

Before preprocessing:

main.cpp

```cpp
#include "add.cpp"

int main()
{
    int a{ 5 };
    int b{ 17 };

    int sum{};
    sum = add(a, b);
}
```

After preprocessing:

main.cpp

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

int main()
{
    int a{ 5 };
    int b{ 17 };

    int sum{};
    sum = add(a, b);
}
```

A brief aside: the top-to-bottom Compiler

A function or variable cannot be *called* before it is *declared*, the compiler will not know what it is yet. *Definition* is allowed to come later.

This is illegal because the add function is called before it is *declared*.

```
int main()
{
    int x{};
    x = add(99, 1);
}

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

However, this is legal.
The add function is *declared* before it is called, and can be *defined* later.

```
// Declaration
int add(int x, int y);

int main()
{
    int sum{};
    sum = add(99, 1); // Call
}

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

This is why header files are useful.
There we can *declare* everything so we can safely call In any order below.
Each function gets a unique signature based on its name, return type and parameters.
These signatures are used to connect function calls to their implementations.

Compilation/Header and Source Files Explained

The compiler uses the following process:

1. Preprocessing – Resolves lines starting with # in each cpp file.
2. Per cpp compilation – Compiles each cpp file individually.
3. Linking – links together functionality in different files.

1. Preprocessing

MyClass.cpp

```cpp
#include "MyClass.h"

MyClass::MyClass() {
    id = 0;
    name = "noname";
};

MyClass::MyClass(std::string name) {
    id = 0;
    this->name = name;
};

void MyClass::Perform() {
    id++;
};

int MyClass::getId() {
    return id;
}

std::string MyClass::getName() {
    return name;
}
```

MyClass.h

```cpp
class MyClass {
private:
    int id;
    std::string name;

public:
    MyClass();
    MyClass(std::string name);
    void Perform();
    int getId();
    std::string getName();
};
```

main.cpp

```cpp
#include <iostream>
#include "MyClass.h"

int main() {
    MyClass objectInst{ MyClass("Joan") };
    std::cout << "MyClass instance's name: "
        << objectInst.getName() << "\n";
}
```

2. cpp Compilation

MyClass.obj

Defines: MyClass variables & funcs

main.obj

Has declarations for MyClass obj

3. Linking

MyClass.obj

Defines: MyClass variables & funcs

main.obj

Has declarations for MyClass obj

```cpp
MyClass::MyClass(std::string name) {
    id = 0;
    this->name = name;
};
```

```cpp
#include <iostream>
#include "MyClass.h"

int main() {
    MyClass objectInst{ MyClass("Joan") };
    std::cout << "MyClass instance's name: "
        << objectInst.getName() << "\n";
}
```

```cpp
std::string MyClass::getName() {
    return name;
```

Vectors

```
void PrintContents(const std::vector<int>& vec)
{
    std::cout << "Length: " << vec.size() << "\tContents: ";
    for (size_t i=0; i < vec.size(); i++)
    {
        std::cout << vec[i] << ", ";
    }
    std::cout << "\n";
}
```

This is the first non-primitive data structure we'll be using in this class.
It is C++'s resizable array. It is a lot more high level and flexible than C style fixed arrays, so it's more useful for us but we should still maintain a low level understanding.

To make a vector we have to provide it with a type to make its elements, this is called a *template*, more on this next week.

```
std::vector<int> emptyVector{};
PrintContents(emptyVector);
std::vector<int> length5Vector( _Count: 5);
PrintContents(length5Vector);
std::vector<int> prefilledVector( _Count: 6,  _Val: 9);
PrintContents(prefilledVector);
```

```
Length: 0    Contents:
Length: 5    Contents: 0, 0, 0, 0, 0,
Length: 6    Contents: 9, 9, 9, 9, 9, 9,
```

Vectors

We can add elements to our vectors in a few ways:

Overwrite a current value:

```
std::vector<int> vec {1,2,3};
PrintContents(vec);
vec[0] = 99;
PrintContents(vec);
```

```
Length: 3    Contents: 1, 2, 3,
Length: 3    Contents: 99, 2, 3,
```

Add an element to the end:

```
vec.push_back(_Val: 4);
PrintContents(vec);
```

```
Length: 4    Contents: 1, 2, 3, 4,
```

More complex ways we can talk about in future weeks

Product of Array Except Self

This exercise is a step up in difficulty from last week's exercises. Friction is learning!

Given a vector, you must return a vector where each element is the product of all elements in apart from the element of the same index.

[1,2,3] => [, ,] => [6, 3, 2]

1*2*3 1*2*3 1*2*3

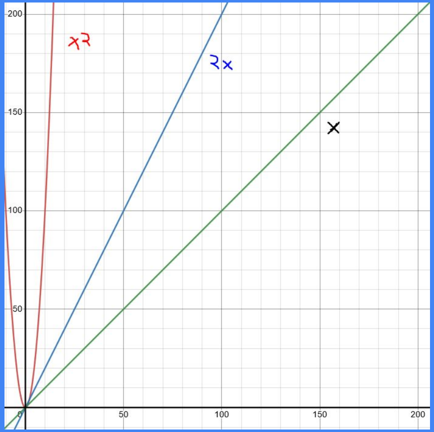
I'll walk you through the straightforward solution to this, and then we can talk about the smarter and faster solution.

O(n^2) and O(n) solutions to AXS

```cpp
std::vector<int> productExceptSelf(const std::vector<int>& nums) {
    std::vector<int> result(nums.size(), 0);

    for (std::size_t i = 0; i < nums.size(); i++) {
        int sum{ 1 };
        for (std::size_t k = 0; k < nums.size(); k++) {
            if (i != k) {
                sum *= nums[k];
            }
        }
        result[i] = sum;
    }
    return result;
}
```

Worst case: O(n^2)
Best case: O(n^2)
Middle case: O(n^2)




```cpp
std::vector<int> productExceptSelf(const std::vector<int>& nums) {
    int numZeros = 0;
    int productWithoutZeros = 1;
    std::size_t zeroIndex = nums.size();
    // compute
    // 1) number of non-zeros
    // 2) product of all non-zeros in nums
    // 3) index of a zero in nums, if there is one
    for (std::size_t i = 0; i < nums.size(); ++i) {
        if (nums[i] == 0) {
            ++numZeros;
            zeroIndex = i;
        }
        else {
            productWithoutZeros *= nums[i];
        }
    }
    // initialise result to be all zero vector
    std::vector<int> result(nums.size());
    if (numZeros == 0) {
        for (std::size_t i = 0; i < nums.size(); ++i) {
            result[i] = productWithoutZeros / nums[i];
        }
        return result;
    }
    // when numZeros == 1, we just have to correct the
    // entry of result corresponding to the 0 in nums
    if (numZeros == 1) {
        result[zeroIndex] = productWithoutZeros;
        return result;
    }
    // when numZeros >= 2, result (= all zero vector) is correct answer
    return result;
}
```

Worst case: O(2*n)
Best case: O(n)
Middle case: Either

Toy Vector

Here we are implementing our own version of the Vector class, to get a better feel for how it works under the hood.

I suggest we implement the functions in this order:

- Constructor that takes an integer input n- Create an all-zero vector of size n
  `MyVector::MyVector(int n)`
- operator`[]` - Get/set the ith element of the vector, as in `vec[0] = 5;`
- Getters for the `size`, `capacity`, and last element of the vector (`back`)
- Destructor- Free the memory allocated for the underlying array. `MyVector::~MyVector()`
- `pop_back` - Remove the last element of the vector.
- `push_back` - Add a new element to the end of the vector. This is the most interesting one. Implement the doubling array scheme that we discussed in lecture.