COMP6771 Week 1.2

Intro & types

A simple look at C++

```cpp
// helloworld.cpp
#include <iostream>

int main() {
  std::cout << "Hello, world!\n";
  return 0;
}
```

Let's break it down:

- Comment
- #include directive
- Standard library elements
- The output stream, cout
- stream insertion operator
- std namespace
- stream manipulator endl
- Return statements
- Semicolons, braces, string literals
- For tutorials: What is similar to C's "scanf" in C++?

Basic compilation of C++

```cpp
// helloworld.cpp
#include <iostream>

int main() {
  std::cout << "Hello, world!\n";
  return 0;
}
```

- For CSE machines:
  - g++ -std=c++17 -o helloworld helloworld.cpp
  - clang++   -std=c++17 -o helloworld helloworld.cpp
- For other, see Webcms3

"\n" vs std::endl

- When you stream something to STDOUT it gives stored in a buffer
- This buffer is eventually "flushed" (e.g. sent to terminal)
- We will use "\n" in this course as it allows more control over when the buffer is flushed, which is important on devices with limitations on performance capabilities.

```
// The following two are equivalent

std::cout << "\n" << std::flush;

std::cout << std::endl;
```

Basic Types

Type	What it stores
bool	True or false
int	Whole numbers
double	Real numbers
char	A single character
string	Text
enum	A single option from a finite, constant set
T*	Raw pointers. Avoid using until we explain when, and **when not**, to use them

There are other basic types, but you will not need them for this course, and will rarely need them in industry.

Type conversion

We will cover this much later in the course.

In the meantime just know that implicit type conversion may happen and not cause any runtime errors

```
bool b1 = 10; // b1 becomes true
bool b2 = 0.0; // b2 becomes false

int i1 = true; // i1 becomes 1
int i2 = false; // i2 becomes 0
```

C++ Operators

Basic C++ operators are very similar to your basic C operators, E.G.

- x.y, x->y, x[y]
- ++x, x++
- x && y, x || y
- And many, many more

Program errors

During the course we will talk about different types of errors:

Compile time

```
int main() {
  // No type specified.
  a = 5;
}
```

Runtime (Exception)

```
#include <string>

int main() {
  std::string s = "";
  s.at(0);
}
```

(Runtime) Undefined behaviour

```
#include <string>

int main() {
  std::string s = "";
  s[0];
}
```

Link time

```
// linker1.cpp
#include <iostream>

int Foo();

int main() {
        std::cout << Foo();
}
```

Runtime (Logic)

```
int main() {
  int a = 3;
  int b = 4;
  int c = 5;
  // Order of operations.
  int average = a + b + c / 3;
}
```

Literals

- Some literals are embedded directly into machine code instructions
- Others are stored in read-only *data* as part of the compiled code

Type of literals	**Examples**
Boolean	true, false
Character	'a', '\n'
Integer	20, 0x14, 20L
Floating-point	12.3, 1.23e4,
String (these are not std::strings)	"Healthy Harold", "a"

Declarations vs Definitions

- A declaration makes known the type and the name of a variable
- A definition is a declaration, but also does extra things
  - A variable definition allocates storage for, and constructs a variable
  - A class definition allows you to create variables of the class' type
  - You can call functions with only a declaration, but must provide a definition later
- Everything must have precisely one definition

```cpp
void DeclaredFn(int arg);
class DeclaredClass;

// This class is defined, but not all the methods are.
class A {
  int DeclaredMethod(double);
  int DefinedMethod(int arg) { return arg; }
}

// These are all defined.
int DefinedFn() { return 1; }
int i;
int j = 1;
std::vector<double> vd;
```

Const

- The value cannot be modified
- Make everything const unless you know it will be modified
    - This course will have a heavy focus on const-correctness

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

int main() {
    const int i = 0; // i is an int
    i++; // not allowed
    std::cout << i << '\n'; // allowed

    const std::vector<int> vec;
    vec[0]; // allowed
    vec[0]++; // not allowed
    vec.push_back(0); // not allowed
}
```

Why const

- Clearer code
  - You can know that a function won't try and modify something just from the signature
  - Immutable objects are easier to reason about
- The compiler **may** be able to make certain optimisations
- Immutable objects are **much** easier to use in multithreading
  - Don't have to worry about race conditions between threads
- Prevents bugs from changing things you shouldn't

References

- We can use pointers in C++ just like C, but generally we don't want to
- A reference is an alias for another object
  - You can use it as you would the original object
- Similar to a pointer, but:
  - Don't need to use -> to access elements
  - Can't be null
  - You can't change what they refer to once set

```cpp
int i = 1;
int j = 2;

int& k = i;
k = j; // This does not make k reference j instead of i. It just changes the valu
std::cout << "i = " << i << ", j = " << j << ", k = " << k << '\n';
```

References to const

- A reference to const means you can't modify the object using the reference
  - The object is still able to be modified, just not through this reference

```cpp
int i = 1;
const int& ref = i;
std::cout << ref << '\n';
i++; // This is fine
std::cout << ref << '\n';
ref++; // This is not


const int j = 1;
const int& jref = j; // this is allowe
int& ref = j; // not allowed
```

Class templates

Type	What it stores	Common usages
std::optional<T>	0 or 1 T's	A function that may fail
std::vector<T>	Any number of T's	Standard "list" type
std::unordered_map<KeyT, ValueT>	Many Key / Value pairs	Standard "hash table" / "map" / "dictionary" type

- Later on, we will introduce a few other types
- There are other types you could use instead of std::vector and std::unordered_map (eg. linked lists), but these are good defaults
- Not a class

How to use class templates

```cpp
#include <unordered_map>
#include <vector>

// The following items are all function DECLARATIONS

// Not allowed - type templates are not types
std::vector GetVector();

// std::vector<int> and std::vector<double> are valid types.
std::vector<int> GetIntVector();
std::vector<double> GetDoubleVector();

// So is combining types
std::vector<std::unordered_map<int, std::string>> GetVectorOfMaps();
```

- These are **NOT** the same as Java's generics, even though they are similar syntax to use
  - std::vector<int> and std::vector<string> are 2 different types (unlike Java, if you're familiar with it)
- We will discuss how this works when we discuss templates in later weeks

Automatic type deduction "auto"

- Let the compiler determine the type for you
- Auto deduction: Take exactly the type on the right-hand side but strip off the **top-level const** and **&**.
  - This is important to know when using auto, to avoid unexpected types

```cpp
auto i = 0; // i is an int

std::vector<int> fn();
auto j = fn(); // j is std::vector<int>

// Pointers
int i;
const int *const p = i;
auto q = p; // const int*
auto const q = p;

// References
const int &i = 1; // int
auto j = i; // int
const auto k = i; // const int
auto &r = i; // const int&
```

Functions

- Breaking code into self-contained functions that perform a single logical operation is one of the backbones of good programming style
- We will cover:
  - The anatomy of a function
  - Default argument values
  - Pass by value
  - Pass by reference

Functions: Default Values

- Functions can use default arguments, which is used if an actual argument is not specified when a function is called
- Default values are used for the *trailing* parameters of a function call - this means that ordering is important

```
string Rgb(short r = 0, short g = 0, short b = 0
Rgb();// rgb(0, 0, 0);
Rgb(100);// Rgb(100, 0, 0);
Rgb(100, 200);      // Rgb(100, 200, 0)
Rgb(100, , 200);    // error
```

Functions

- Formal parameters: Those that appear in function definition
- Actual parameters (arguments): Those that appear when calling the function
- Function must specify a return type, which may be void

```
foo(int bar); // not ok
int foo(int bar); // OK
void bar(); // OK
```

Functions: Pass by value

- The actual argument is copied into the memory being used to hold the formal parameters value during the function call/execution

```cpp
#include <iostream>

void swap(int x, int y) {
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
}

int main() {
  int i = 1, j = 2;
  std::cout << i << " " << j << std::endl;
  swap(i, j);
  std::cout << i << " " << j << std::endl;
}
```

```cpp
#include <iostream>

void swap(int *x, int *y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

int main() {
  int i = 1, j = 2;
  std::cout << i << " " << j << std::endl;
  swap(&i, &j);
  std::cout << i << " " << j << std::endl;
}
```

Functions: Pass by reference

- The formal parameter merely acts as an alias for the actual parameter
- Anytime the method/function uses the formal parameter (for reading or writing), it is actually using the actual parameter
- Pass by reference is useful when:
  - The argument has no copy operation
  - The argument is large

```cpp
#include <iostream>

void swap(int& x, int& y) {
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
}

int main() {
  int i = 1, j = 2;
  std::cout << i << " " << j << std::endl;
  swap(i, j);
  std::cout << i << " " << j << std::endl;
}
```

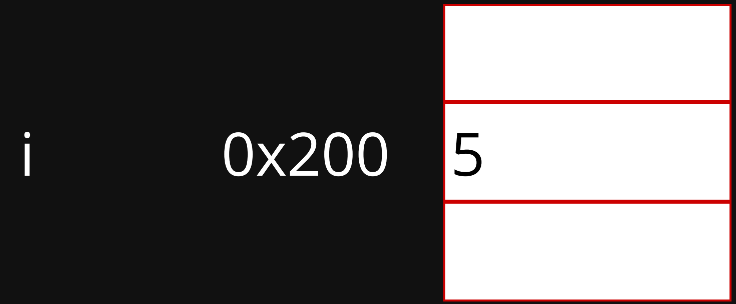
```cpp
void swap(int i, int j);   // 1st-year style
void swap(int& i, int& j); // C++ style

// Note that C does not support
// pass-by-reference. This is pass-by-value.
// C courses often call this
// pass-by-reference because this is
// the closest C has to it.
void swap(int* i, int* j); // C style
```

Lvalue and Rvalue

- Understanding whether an item is an **l**value or an **r**value can help you better understand why particular code expressions do or do not work.
- We will cover examples of what **l**values or **r**values are

Lvalue and Rvalue

```
int i = 5;
i = i + 1;
```



- Add the **rvalue 5 and 1** and store **6** into **lvalue 0x200**
- Simplified thinking:
  - **rvalues** may only appear on the RHS of an assignment
  - **lvalues** may appear on both the LHS and RHS

Lvalue and Rvalue

- Call by value:
  - The **rvalue** of an actual argument is passed
  - Cannot access/modify the actual argument in the callee
- Call by reference:
  - The **lvalue** of an actual argument is passed
  - May access/modify directly the actual argument
  - Eliminates the overhead of passing a large object

Function Overloading

- Function overloading refers to a family of functions in the **same scope** that have the **same name** but **different formal parameters**.
- This can make code easier to write and understand

```cpp
void Print(double d);        // (1)
int  Print(std::string s); // (2)
void Print(char c);          // (3)
Print(3.14);                 // call (1)
Print("hello World!");       // call (2)
Print('A');                  // call (3)
```

Overload Resolution

- This is the process of "function matching"
- Step 1: Find candidate functions: Same name
- Step 2: Select viable ones: Same number arguments + each argument convertible
- Step 3: Find a best-match: Type much better in at least one argument

Errors in function matching are found during compile time

Return types are ignored. Read more about this here.

```cpp
void G();
void F(int);
void F(int, int);
void F(double, double = 3.14);
F(5.6); // calls f(double, double)
```

- When writing code, try and only create overloads that are trivial
  - If non-trivial to understand, name your functions differently

Function overloading and const

When doing **call by value**, top-level const has no effect on the objects passed ot the function. A parameter that has a top-level const is indistinguishable from the one without

```
// Top-level const ignored
Record Lookup(Phone p);
Record Lookup(const Phone p); // redefinition

// Low-level const not ignored
Record Lookup(Phone &p); (1)
Record Lookup(const Phone &p); (2)

Phone p;
const Phone q;
Lookup(p); // (1)
Lookup(q); // (2)
```

Constexpr

- Either:
  - A variable that can be calculated at compile time
  - A function that, if its inputs are known at compile time, can be run at compile time

```cpp
// Beats a #define any day.
constexpr int max_n = 10;

// This can be called at compile time, or at runtime
constexpr int ConstexprFactorial(int n) {
  return n <= 1 ? 1 : n * ConstexprFactorial(n - 1);
}
constexpr int tenfactorial = ConstexprFactorial(10);

// This may not be called at compile time
int Factorial(int n) {
  return n <= 1 ? 1 : n * Factorial(n - 1);
}
// This will fail to compile
constexpr int ninefactorial = Factorial(9);
```

Constexpr (Benefits)

- Benefits:
  - Values that can be determined at compile time mean less processing is needed at runtime, resulting in an overall faster program execution
  - Shifts potential sources of errors to compile time instead of runtime (easier to debug)

Matt: Boring legal stuff

- I'm required to disclose that I work at Google
  - Nothing I say during this course represents Google
  - I am working as a lecturer independently of my work at Google
  - We are using Google products throughout this course, but they are open source

Header files

- Place declarations in header files, and definitions in cpp files
- But we never mentioned hello_world.cpp. How does the compiler know that we meant that one?

```cpp
// path/to/hello_world.h
#ifndef HELLO_WORLD_H
#define HELLO_WORLD_H

void HelloWorld();

#endif // HELLO_WORLD_H
```

```cpp
// path/to/hello_world.cpp

#include "path/to/hello_world.h"

#include <iostream>

void HelloWorld() {
  std::cout << "Hello world\n";
}
```

```cpp
// main.cpp
#include "path/to/hello_world.h"

int main() {
  helloWorld();
}
```

The linker

g++ -c printer.cpp -Wall -Werror -std=c++17 -<more args>

g++ -c hello_world.cpp -Wall -Werror -std=c++17 -<more args>

g++ main.cpp hello_world.o printer.o -Wall -Werror -std=c++17 -<more args>

```
// path/to/hello_world.h
#ifndef HELLO_WORLD_H
#define HELLO_WORLD_H

void HelloWorld();

#endif // HELLO_WORLD_H
```

```
// path/to/printer.h
#ifndef PRINTER_H
#define PRINTER_H

#include <string>

void Print(std::string);

#endif // PRINTER_H
```

```
// path/to/binary/main.cpp
#include "path/to/hello_world.
 
int main() {
  helloWorld();
}
```

```
// path/to/hello_world.cpp

#include "path/to/hello_world.h"
#include "path/to/printer.h"

void HelloWorld() {
  print("Hello world");
}
```

```
// path/to/printer.cpp

#include "path/to/printer.h"

#include <iostream>
#include <string>

void Print(std::string s) {
  std::cout << s << '\n';
}
```

The problem

- Imagine having thousands of header and cpp files?
- You have a few options
  - Manually create each library and make sure you link all the dependencies
    - You would have to make sure you linked them all in the right order
  - Create one massive binary and give it all the headers and cpp files
    - Extremely slow
    - Hard to build just parts of the code (eg. To run tests on one file)
  - Makefiles
    - Unwieldy at large scale (hard to read and hard to write)
  - **Any better options?**

The solution - build systems

- We will be using bazel
- Works out what compilation commands to run for you
- Run using "bazel run //path/to/binary:main"
- Compile a single component using "bazel build //path/to:hello_world"
- We will show you how to do testing next week - it's really easy

```
// path/to/BUILD

cc_library(
  name = "hello_world",
  srcs = ["hello_world.cpp"],
  hdrs = ["hello_world.h"],
  deps = []
)

cc_library(
  name = "printer",
  srcs = ["printer.cpp"]
  hdrs = ["printer.h"],
  deps = [
    # If it's declared within the same build
    # file, we can skip the directory
    ":hello_world"
  ]
)
```

```
// path/to/binary/BUILD

cc_binary(
  name = "main"
  srcs = ["main.cpp"],
  deps = [
    "//path/to:hello_world"
  ]
)
```

BUILD files

- One build file per folder
- Each build file has multiple rules

Build rules

- Each build rule has:
  - A name
  - A list of sources (srcs)
  - A list of headers (hdrs)
  - A list of dependencies (dep)
  - Potentially many other arguments (https://docs.bazel.build/versions/master/be/c-cpp.html#cc_library)
- Refer to build rule by //path/to/dir:<name>

Types of build rules

- cc_library
  - A piece of code that can't run on its own, but can be depended upon by other files
- cc_binary
  - The srcs should have a main function
  - Has no headers
  - Cannot be tested
- cc_test
  - Works very similar to a binary
  - Semantic difference

Common mistakes

- When running code with bazel, it runs in a different directory to the source
  - If you try and open files, this will fail (you need a data param)
- You may get an error saying that a #include isn't found
  - You're probably missing a dependency in your BUILD rule
- You may not have the debug option available in CLion
  - When you add build rules, CLion won't know about them till it updates
  - Run bazel > sync

Setting up your computer

- The officially supported way will be to use the VM provided
  - Install virtualbox and download the VM at http://tiny.cc/comp6771vm
  - Create a jetbrains account with your student email address (https://www.jetbrains.com/shop/eform/students)
  - File > import appliance > 6771.ova (the downloaded file)
    - **Make sure you modify the RAM and CPU you give the VM**
  - When you start up the VM, start up clion, and log in with that account
- Feel free to try and set it up on your own computer without the VM (see the README in the course repository), **but we will not help you**
  - If you get it to work, please send me a pull request with the steps required to add to the README