## Context

The course will be using C++ 17.

Modern C++ is powerful and fast. It can be difficult, but it doesn’t have to be. Modern C++ is clean and simple, has differences to old C++, but the standard library does a lot of the work for us now.

ISO Standards committee official page: isocpp.org

Reference page: en.cppreference.com/w/

## Tools

**A compiler and a linker:**

The **Building** process is made up of compilation and linking. Source files **.cpp** are compiled to **.o/.obj** depending on the system. The object files will then be linked together to create the exe. Why are two separate processes required then? Because when singular files are changed in a massive system, this allows only the changed files to be recompiled and linked to the others, increasing build speed for each cycle. Imported libraries are also never changed, unless updated, so they will only be compiled once.

A context-aware text editor with documentation and intellisense. (Visual Studio Community 2022)

Debugger

**Popular tools:**

* Compiler
  + Clang
  + GCC
* IDE: (Integrated Development Environment)
  + Visual Studio
    - Community edition – free for all purposes
    - Professional – for big companies
    - Enterprise – additional features
  + Visual Studio Code (lacks a built-in compiler and requires console building)
  + CLion
* Online compilers:
  + Great for comparing compilers
  + Testing new features on compilers, which support them
  + Sharing snippets

**Kinds of applications**

C++ is general-purpose, so it can be used for building programs on all kinds of platforms.

* Console applications – **Used for the course**
* PC apps – Windows/Linux/Mac
* Phone games and apps
* Etc…

**Making a project into Visual Studio**

When a new project is created depending on the type Visual Studio will include precompiled headers. Which are okay for large projects, but here are not needed.

**Variables and Fundamental Types**

C++ is a strongly typed language, which means that a variable’s type is set before its first use and cannot be changed after. You can use casting to convert types and make your intentions clear in code. This is useful for numeric conversions – int to double etc, or casting to an expected subclass of a parent. There are **safe casts**.

y = static\_cast<int>(x);

Fundamental types are made up of **Booleans, numeric types and character types**, with these three also known as arithmetic types, and **void and nullptr.** User defined types, or **compound** types, are strings, business objects and anything else made up of other fundamental and compound types. **Auto** can be used to tell the compiler to deduce the type. This is used for readability for example, when setting a variable’s value to the return value of a function, meaning that the compiler can safely deduce the correct type. In this way, C++ is still a strongly typed language. Useful for ugly declarations.

Can use digital separators in **int** for readability – 1**’**000**’**000.  
0**x**1231 – Hex ints  
0**b**111 – Bool ints 😊

Setting fundamental types to a value outside of their range will be automatically converted from the bit pattern to an appropriate number in the range aka Overflow and Underflow.

**Building**

the #include directive copies the contents of a **.h** header file. Header files only outline what functions there are and what types they return or take as parameters. They are the outline, which other files then **#include** for use in their own implementation, which is done inside the **.cpp** C++ files, which in turn implement the outlines of the header file. When the linker combines the implementations of each compiled file, it makes sure that each function or variable is available for calling.

**User defined types (Classes and Objects)**

Class declaration syntax:

*class* **Name{**  
 private: // outline private stuff (also the default declaration if none given  
 std::string someString;  
 public: // outline public stuff

**};**

Then in the .cpp file you **#include Name.h** and implement the functions.

Objects are **instances** of a **class**. Objects are **initialized** with a **constructor** by passing arguments for its fields or using default values. The **constructor** has the same name as the class and any number of arguments. When an object is created, all its fields are initialized using default constructors. By using our object constructor, we gain control over that first initialization and dismiss having to overwrite those values in a different function or a setter. aka optimization and proper practice. Default constructors exist for objects when one is not explicitly written. You can define a default constructor more readably by using **Name()**  = default; in the header file. A default constructor is considered *“a constructor that doesn’t take any parameters*”

Create objects have a lifetime, called **scope**. This is the time it exists on the **stack** and takes up memory and is generally between its constructor and **destructor**, which allows clearing up resources in a proper manner. C++ scopes are defined by its surrounding flow control symbols – aka objects can exists in the scope of a flow statement – ifs and fors, in the scope of an object, a request etc, whatever is required, or they can be global, which means they exist the whole time the program runs.

**Struct**

Besides classes there is another keyword used to denote “plain old data” or a class without business logic **– struct­.** A structure can have functions, constructors, and destructors like a normal class, but everything not explicitly defined using an access modifier is **public** instead or private.

**//**A struct or structure defines a container of objects without any functionality attached to them.

**Namespaces**

They are made to prevent name collisions so we can have our own *string* class, without the compiler complaining about a collision with the standard library’s *string* class. The syntax is **namespace::field/function**. Namespaces can be used for whole classes to avoid writing **namespace::** on the beginning of each call by writing special syntax after our directives. This can be done on the whole namespace or only functions/fields from it. Even if using the full syntax, **never** use using statements in header files.

|  |  |
| --- | --- |
| using namespace **something;** | using namespace **std;** |
| using **something;** | using **std::string;** |

**Inheritance**

Classes in C++ can be inherited. This allows the subclass to inherit everything its parent has (besides fields with a private modifier) and add/override its own functionality without creating a whole new class. We only specify new fields and constructor behaviour. Since C++ supports multiple-inheritance, we call base constructors by their specific name (e.g. Animal(params…) instead of super(params…)).

class Dog : <- colons denote inheritance  
 public Animal

{  
 <- only specify new fields

};

Inherited classes usually have a build pattern of construction from base to subclass and destruction the other way around. Be careful when casting subclasses to base classes not to delete them as that will cause a memory leak. Maybe?

**Enum**

Enumerations is a way of given names to a set of constants. Useful for consistent values and good programming practices, but once you have a named value of one enum, you can’t use it in another one. To avoid name collision C++ has no added scoped enums, also called **enum class**, which allow this by using the enums fully qualified name. Calling is the same syntax as namespaces – **PlayerState::alive**.

So if we have an:  
**enum PlayerState**

{  
 alive,  
 dead  
}

**alive** and **dead** cant be used in another enum, but we can have them duplicated like this:

|  |  |
| --- | --- |
| **enum PlayerState**  {  alive,  dead } | **enum EnemyState**  {  alive,  dead, } |

Ideally these would be part of the same **CharacterState** enum, but that is besides the point.

**Preprocessor**

The preprocessor takes place in **translation phase 4**. The result of preprocessing is a single file, which is then passed to the actual compiler. Preprocessor commands take the following syntax:   
**# command *arguments***.

Since include directives basically paste the chosen content to the file, there may be conflicts such as class redefinitions. To avoid this, we use the directive **#pragma once** to . Its technically not a standard directive, but all major compilers support it.

## Flow Control

This section deals with control of the program flow when things don’t happen linearly aka conditions, repeated actions, calculations and more. This course will be cover **if, for, break, continue, immediate if(ternary)**. Specifics:

* brackets are optional, but only cover the next command. Generally brackets are used for readability regardless.
* can exit loops earlier with **break** if a condition is met.

## Functions

Functions in C++ do **NOT** have to be part of a class, those are called **free functions**. Functions are usually pass-by-value, meaning the function will modify a copy of the given variable, but it can also be pass-by-reference, where the given variable is the one being modified and you don’t require a return statement for the result. The syntax for both is slightly different, but there will be more context on it in the **indirection** chapter later. Syntax:

int someFunction(**int** x) - Value  
int someFunction(**int&** x) - Reference

We generally don’t want to pass references for arithmetic types, but if we do need to, we should use **const** references, meaning the value will never change, and can’t be modified in the function itself. This guarantees data integrity.

**Member functions** are part of a class and usually represent his behaviour intended for use in-class or outside. Functions can be **const**, meaning it does not modify any of its member fields. You can declare inline member functions in header files if they are simple and short e.g. getter/setters.

**Error messages** tell you what has gone wrong where and they usually come from two main places – the compiler and the linker. The compiler errors tell us we haven’t made any promises about something, while the linker cares about keeping that promise. The compiler will require used functions be declared in header files, while the linker will try to find their implementations and cry if it doesn’t find it.

## Operators

|  |  |
| --- | --- |
| Arithmetic operators | + - \* / |
| Shortcut arithmetic | += -= \*= /= |
| Increment operators | i++ ++i |
| Modulo operator | % |
| Comparison | == != |
| Boolean | && || |