Lecture 1: Intro to C++

Language Design Objectives

General aims (§4.2)

- C++'s evolution must be **driven by real problems**
- **don't** get involved in a **sterile quest** for **perfection**
- C++ must be useful **now**
- **every feature** must have a **reasonably obvious implementation**
- always **provide a transition** path
- C++ is a **language, not a complete system**
- provide **comprehensive support** for each **supported style**
- **don't** try to **force people** (to use constructs they don't want or need)

Design Support Rules (§4.3)

- support **sound** design notions
- provide **facilities** for **program organization**
- **say** what you **mean**
- **all features** must be **affordable**
- it's **more important** to **allow a useful feature** than to **prevent every misuse**
- support **composition** of software from **separately developed parts**

Language-technical Rules (§4.4)

- **no implicit violations** of the **static type system**
- provide **as good support** for **user-defined** types as for **built-in types**
- **locality** is good
- **avoid order dependencies**
- **if in doubt**, pick the variant of a feature that is the **easiest to teach**
- **syntax matters** (often in perverse ways)
- **preprocessor usage** should be **eliminated**

Low-level Programming Support Rules (§4.5)

- **use traditional** (dumb) **linkers**
- **no gratuitous incompatibilities** with C
- **leave no room** for a **lower-level language** below C++ (except assembler)
- what you **don't use**, you **don't pay for** (zero-overhead rule)
- **if in doubt**, provide means for **manual control**

C++ Today

- compatible with almost everything from the **C language**
- full **static-type checking**
- ability to **overload functions** including for opeartors, eg: ==, <, ++, ()

- **generic programming** and **static polymorphism**
- **object-oriented programming** and **dynamic polymorphism**
  - encapsulation, inheritance, virtual member functions in addition to references, data, and function pointers
- exception handling; namespaces
- value, pointer, and reference (lvalue and revalue) semantics
- **concurrency** and its associated semantics
- C standard library and C++ standard library

Some Important Terms

The following are some **importnant concepts and termms** in C++

- namespaces
- `struct, class`, and `union`
- exception safety (no exception, basic exception, strong exception safties, and no-throw guarantee)
- opaque pointers and forward declarations
- header files, header-only libraries
- translation unit
- static and dynamic linking; internal and external linkage
- "plain old data" (POD)
- value, reference, and pointer semantics; lvalues and rvalues; swapping, copying, moving
- One Definition Rule (ODR)
- Resource Acquisition is Initialization (RAII)
- non-virtual (tree) and virtual (graph) inheritance
- non-virtual and virtual member functions
- cv-qualifiers (ie `const, volatile`) and `constexpr`
- sequence point, threads, locks, futures, promises
- templates; traits / policy classes; CRTP; SFINAE
- unspecified, implementation-defined, and undefined behaviour
- static, thread, automatic, dynamic storage notions; in-place (non-dynamic) object creation and destruction

Programming Paradigms

Imperative Paradigm

Defined computation in terms of programming statements that describe changes in state (ie. program statements describe *how* something is to be done instead of describing only *what* is to be done. The imperative paradigm includes the **procedural** paradigm.

Procedural Paradigm

> "Decide which procedures you want; use the best algorithms you can find" [§2.3, p23]
>
> "The idea of composing a program out of functions operating on arguments. Explicit abstraction mechanisms are not uses" [§22.1.3, p781]

The **procedural** paradigm is an **imperative** paradigm originally represented by the C language subset, providing:

- set of **scalar data types**
- ability to define **arrays, aggregate data types**
- set of **conditional constructs, looping constructs, and built-in operators**
- ability to define, reference, and call **functions**
- changing the values of **variables**
- ability to query, set, and manipulate **addresses**

Modular Paradigm

> "Decide which modules you want; partition the program so data is hidden within modules" [5, §2.4, p26]
> "Compose our systems out of 'components' that we can build, understand, and test in isolation. […] so that they can be used in more than one program ('reused')" [4, §22.1.2.5, p779

Modular paradigm enables **encapsulation**: the ability to expose or hide functions/types defined in different modules. They provide a simple but limited way to abstract data

- in C++, module can be defined by the contents of a source file, or, "translation unit"
- in C++, module can also be defined by a `namespace`s, `struct`s, `union`s, or `class`es

Data Abstraction

> "The idea of first providing a set of types suitable for an application area and then writing the program using those. This is called 'abstraction' because a type is used through an interface" [4, §22.1.3, p781]
>
> "Decide which types you want; provide a full set of operations for each type" [5, §2.5.2, p32]

Object-Based Paradigm

Object-based paradigm employs data structures called "objects" that are composed of state (ie. represented by variables) and methods where object state is used to determine what is executed next.
eg: in C++, using `struct` or `class` or objects represented as modules is object-based

Object-Oriented Paradigm

> "The idea of organizing types into hierarchies to express their relationships directly in code" [4, §22.1.3, p781]
>
> "Device which class you want; provide a full set of operations for each class; make commonality explicit by using inheritance" [5, §2.6.2, p39]

Object-oriented paradigm is object-based programming with **inheritance** added. Inheritance enables expressing **hierarchical relationships** directly in code.

Generic Paradigm

> "Decide which algorithms you want; parameterize them so that they work for a variety of suitable types and data structures" [5, §2.5.2, p32]

- In C++, generics enable powerful forms of **compile-time polymorphism**; exploiting them relies focusing on **algorithms** and **type abstractions** as **patterns**.
- **Generic** paradigm allows code to be written using special placeholders representing as-yet unknown **types** and **constant values**. These placeholders are called **parameters**.
- Generic code is turned into real code by **substituting** real types and constant values for each parameter. This process is called **template instantiation**
- C++ is **statically-typed**. Any and all use of generics **must be fully evaluated and determined at compile-time**
- in C++, generic use requires defining and using `template`s. Just as functions have arguments, templates have parameters. However, function args are substituted at **run-time** whereas template params are **instantiated** at **compile-time**

Template Metaprogramming

Template metaprogramming paradigm is a generic paradigm where the representationo of the generics is **Turing-complete** and such is being **exploited**

- **any computation** that can be computed by a **computer program** can be performed using template metaprogramming
- templates **evaluated at compile-time** - not run-time!

Functional Paradigm

Functional paradigm describes the evaluation of code using **pure functions**

- template metaprogramming (and C++11's `constexpr`) functions utilize the functional paradigm
- `<type_traits>` header provides a number of template metaprogramming operations

Multiparadigm Programming Language

A language which allows one to freely **mix** multiple programming paradigms is a multiparadigm programming language. C++ is one of these languages.

First Steps

Usage of C library header files requires dropping .h and prepending a 'c' to the header file. eg: `<stdio.h>` ⇒ `<cstdio>`

Using the std Namespace

- we'll start off by writing `using namespace std;` at *file scope* after all `include`s. This tells the compiler to search the std namespace for all unresolved symbols. We can avoid explicitly writing `std::` in from of all std namespace symbols
- next, we can move `using namespace std;` to smaller scopes such as inside functions. This helps reduce ambiguous symbol compiler errors
- finally, we use instead tell the compiler to where to look for specific symbols explicitly: `using std::cout;`

Default Streams

C++ defines the following always-open stream objects (instances of `std::istream` and `std::ostream` classes) in the std namespaces:

Stream	Header	C Equivalent	Remarks
`cin`	`<istream>`	`stdin`	Standard input
`cout`	`<ostream>`	`stdout`	Standard output (buffered)
`cerr`	`<ostream>`	`stderr`	Standard error (unbuffered)
`clog`	`<ostream>`	`stderr`	Standard error (buffered)

Using Stream Operators (Reading and Writing)

Stream classes can be used by overloading the bit-shift operators:

- use `operator >>` to read data from a stream
- use `operator <<` to write data to a stream

I/O for User-Defined Types

Suppose we have a type called `MYTYPE`, then we can read it in by writing:

```
#include <istream>
std::istream& operator >>(std::istream& is, MYTYPES& t)
```

```
{
    // Code to read in type here...
    is >> t.some_attribute;
    return is;
}
```

.. and write it out with:

```
#include <ostream>
std::ostream& operator <<(std::ostream& os, MYTYPE const& t)
{
    // Code to write out type here
    os << t.some_attribute;
    return os;
}
```

Error Detection for Streams

If a stream is implicitly or explicitly **cast** to a `bool`, the result is:

- `true` iff there have been no errors on the stream
- `false` otherwise

If **any** error occurs on a stream in C++, **no furthuer I/O** occurs on that stream **until** the error is cleared. The stream errors are:

ios_base Constant	Member Function	Remarks
`goodbit`	`good()`	I/O works. Not an error.
`eofbit`	`eof()`	The end-of-file was encountered.
`badbit`	`bad()`	An operation failed an is unrecoverable.
`failbit`	`fail()`	An operation failed and is possible recoverable.

Compiling C++ Code

Whether using GNU GCC's g++ compiler or LLVM clang++, use these opetions:

ISO Level	g++/clang++ cli options
C++98	`-std=c++98 -Wall -Wextra -Werror -Wold-style-cast file.cxx`
C++11	`-std=c++11 -Wall -Wextra -Werror -Wold-style-cast file.cxx`
C++14	`-std=c++14 -Wall -Wextra -Werror -Wold-style-cast file.cxx`
C++17	`-std=c++17 -Wall -Wextra -Werror -Wold-style-cast file.cxx`

- if using **C++ threads**, use `-pthread` option
- if using **GDB**, use `--ggdb` option