C++ 01 - Introduction to the C++ language

Feb'22

EPITA Research & Development Laboratory (LRDE)

Introduction

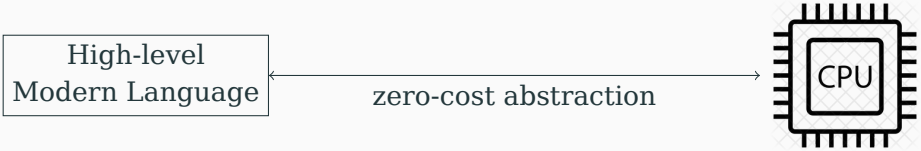
Building your first C++ program

Ecosystem & Build system

Introduction

What is C++

Language for solving **practical**, **industrial** real-world problems
(*not* academical or research language)

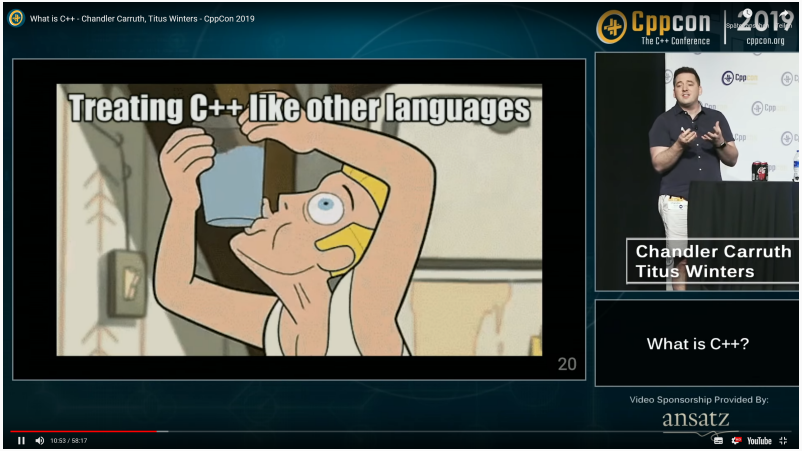


- High Level Abstraction
- Multi-paradigm *procedural,object-oriented,function*
- Modern eco-system *modules, package manager*
- Modern features *async code, lambda functions*
- Portability & Stability

Performance

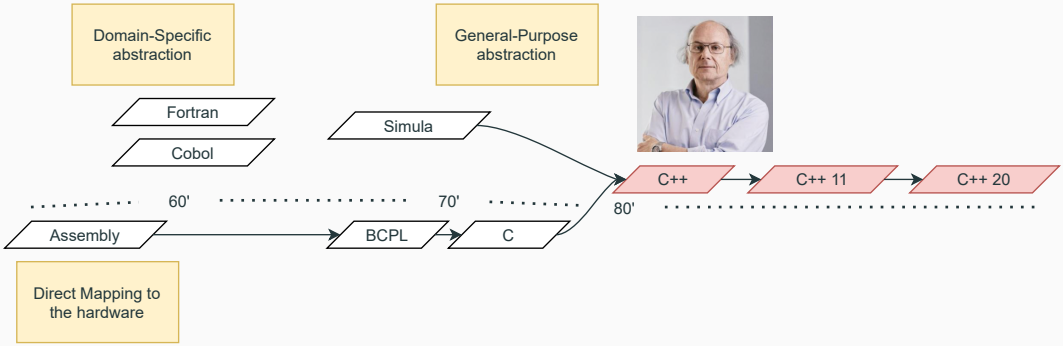
- Maps directly to the hardware (instructions and native types)
- Access to low-level feature of the Abstract Machine

What is C++

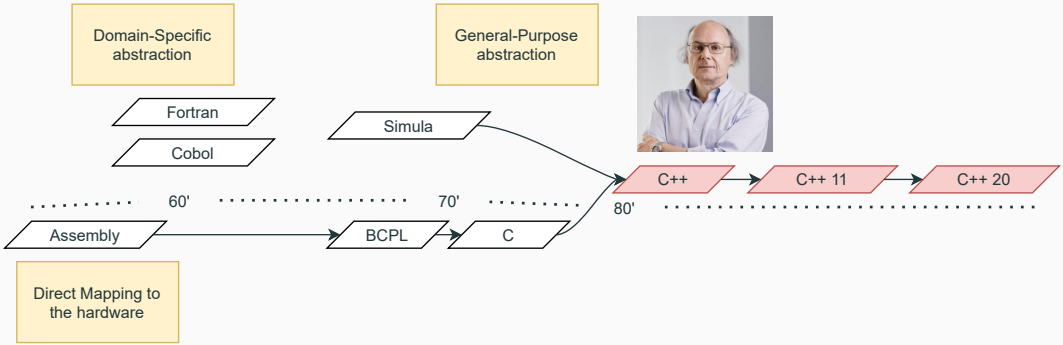


https://youtu.be/LJh5QCV4wDg

The history of C++



The history of C++

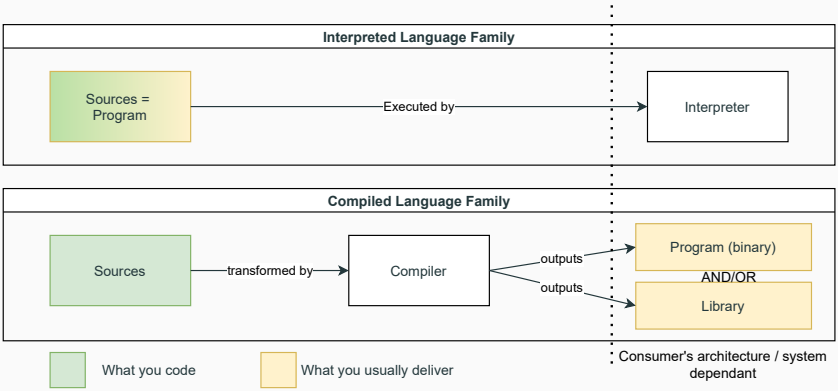


Objectives of this course

- An introduction to C++
- An idea why C++ is different
- An idea why C++ is fast
- An idea why C++ is popular

Building your first C++ program

C++ is compiled

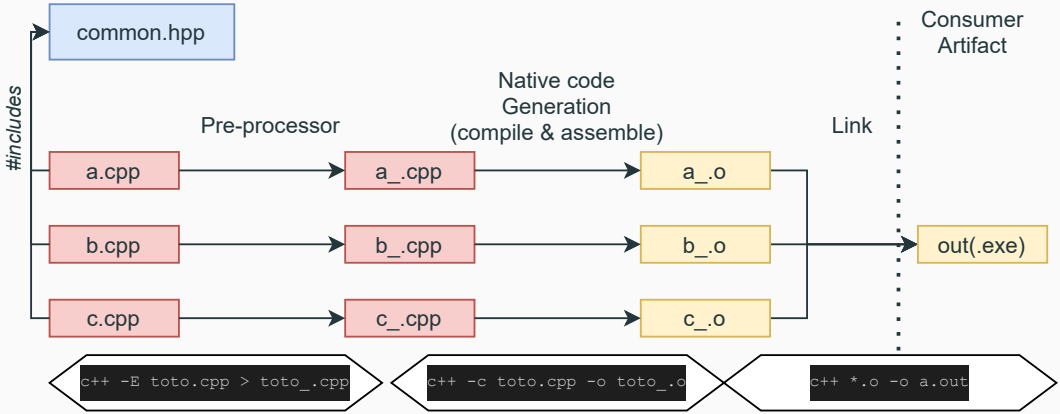


Note that the final artifact has generally two forms:

- A pure program (executable)
- A library

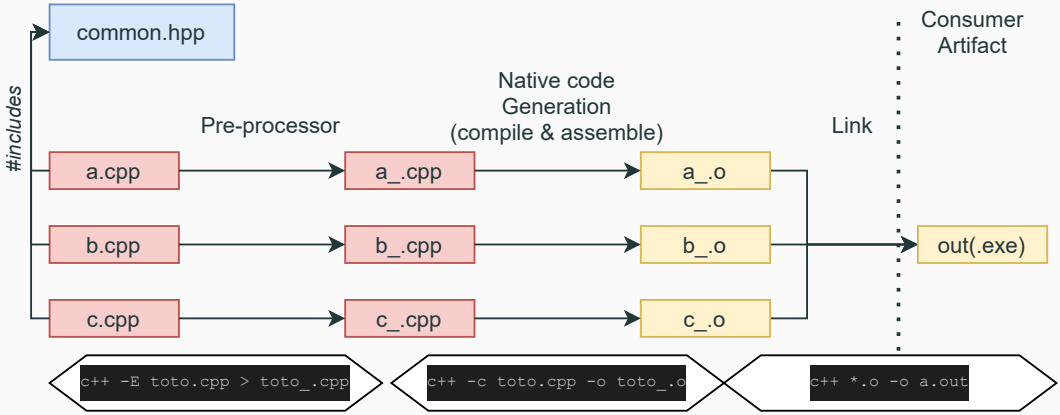
This makes a huge difference: the compilation model **and** the type artifacts influences the source code separation and organization !

Compilation chain overview (program)



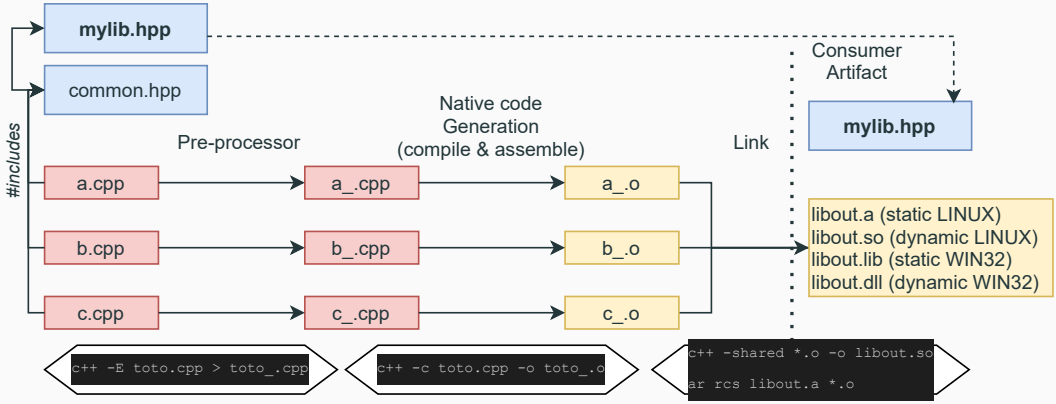
Source code usually split over multiple files.

- The **headers** contain features used jointly by several `cpp`
- The **cpp** files contain the implementation
  1 `cpp` ↔ 1 Translation Unit (TU) ↔ 1 object file `.o`
- The linker joins the object files



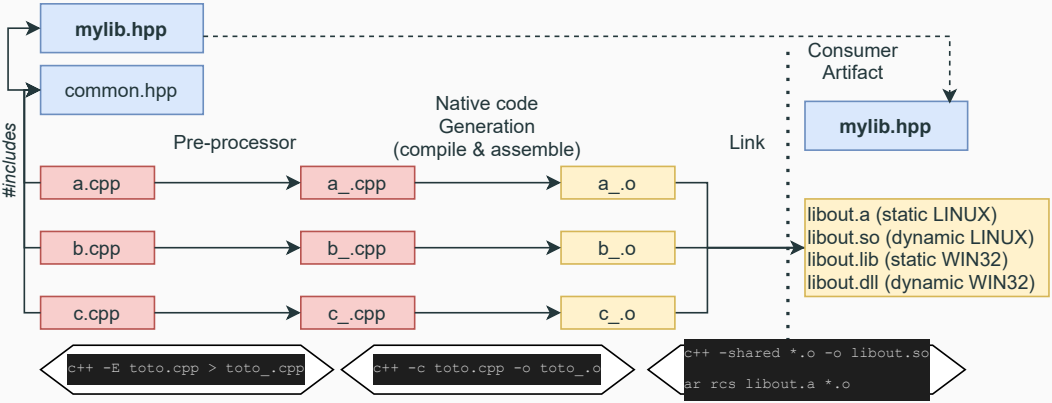
- The C/C++ compilation model compiles each TU independently (distributed/parallel compilation 👍)
- All in one: `c++ -o out[.exe] a.cpp b.cpp c.cpp`
- `C++ 20 Modules` are going to be a game changer on this model

Compilation chain overview (library)



Same principle, but you provide **headers** that form the API (Application Public Interface) of your library to be usable.

Compilation chain overview (library)



Same principle, but you provide **headers** that form the API (Application Public Interface) of your library to be usable.

So what goes in the `cpp` and what goes in the `hpp` ?

Reminder about declaration and definition

Declaration only

```
extern int i;     // Int
int square(int); // Function
struct Foo;       // Struct
enum Color;
```

Decl. + **Definition** *(implementation)*

```
int j;
extern int i = 3;
int square(int x) { return x*x;}
struct Foo { int k; };
enum Color { RED, GREEN, BLUE};
```

- Most of the time, declaring a symbol is enough to use it in your code
- It can be defined in another Translation Unit.

One Definition Rule (ODR)

A symbol can only be defined once (in a translation unit), but it can be declared multiple times.

Reminder about declaration and definition

Declaration only

```
extern int i;     // Int
int square(int); // Function
struct Foo;       // Struct
enum Color;
```

Decl. + **Definition** *(implementation)*

```
int j;
extern int i = 3;
int square(int x) { return x*x;}
struct Foo { int k; };
enum Color { RED, GREEN, BLUE};
```

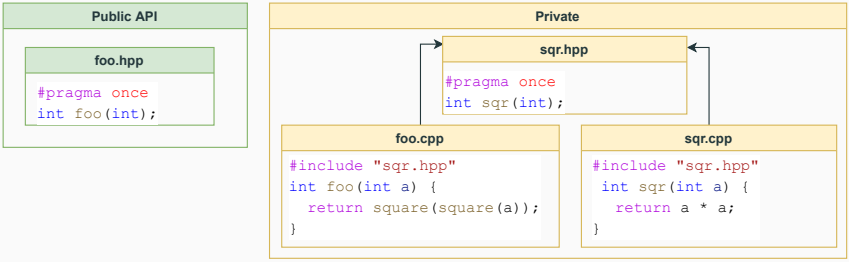
- Most of the time, declaring a symbol is enough to use it in your code
- It can be defined in another Translation Unit.

One Definition Rule (ODR)

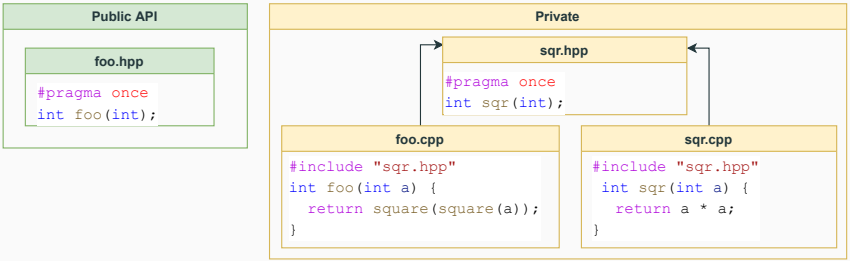
A symbol can only be defined once (in a translation unit), but it can be declared multiple times.

☞ `extern` is a *linkage* instruction that tells the linker that the symbol can be used or defined by other TU (the contrary is `static`)

Splitting the code



- *Public headers* contain *mostly* declaration of the API features
- *Private headers* contain *mostly* declaration of the private features common to several TU
- *Source files* contain the definitions
- `#include`: (text-based) preprocessing directive, it copy-pastes the content
- `#pragma once`: avoids multiple includes of the same header (in case of `#include` chain)



Path	Files
`libfoo/include/*`	Public API (.hpp)
`libfoo/src/*`	Private .hpp + .cpp

Ecosystem & Build system

Ecosystem

- Main compilers: GCC, Clang, MSVC
- Build Systems: Make, Ninja, Bazel, build2
- Build System Generators: **CMake**, autotools
- Package Managers: **Conan**, vcpkg
- Debugging: GDB, LLDB, Mozilla's RR
- Codeformatting: ClangFormat
- Testing: CATCH2, BOOST.TEST, GOOGLE TEST, CUTE
- Sanitizers: Help you find leaks and undefined behaviour
- Leak checkers: Valgrind, Deleaker, HeapTrack
- Static-Analysis: Clang-tidy/Clang-analyzer
- Powerfull IDEs

CMake Build System

CMake is an open-source, cross-platform family of tools designed to build, test and package software.

CMake allows you to:

- define the executables and libraries
- platform dependent compilation options
- provides a “high-level” API

Minimal CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(my_project)
set(CMAKE_CXX_STANDARD 20)      # Project wide C++ standard
add_compile_options(-Wall -Wextra -Werror) # Project wide coding standard

add_executable(main main.cpp)
```