

Clase GPU 13/04/2021

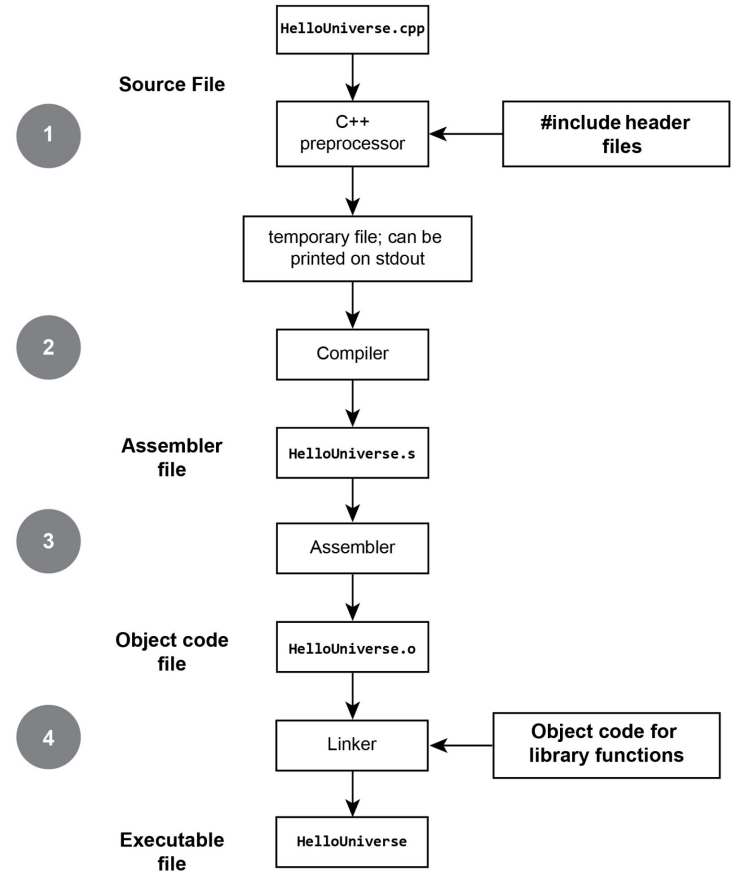
Prof: Nancy Hitschfeld
Prof Aux: Pablo Pizarro
Ayudantes: Pablo Helguero
Heinich Porro

Compilando C++ con CMake

- Modelo de compilación en C++
- Build systems
- ¿Por qué CMake?
- Estructura típica de un proyecto en C++/CMake
- Conceptos básicos de CMake moderno
- Ejemplo: Integrar librerías de testing en un proyecto C++/CMake

Modelo de compilación en C++

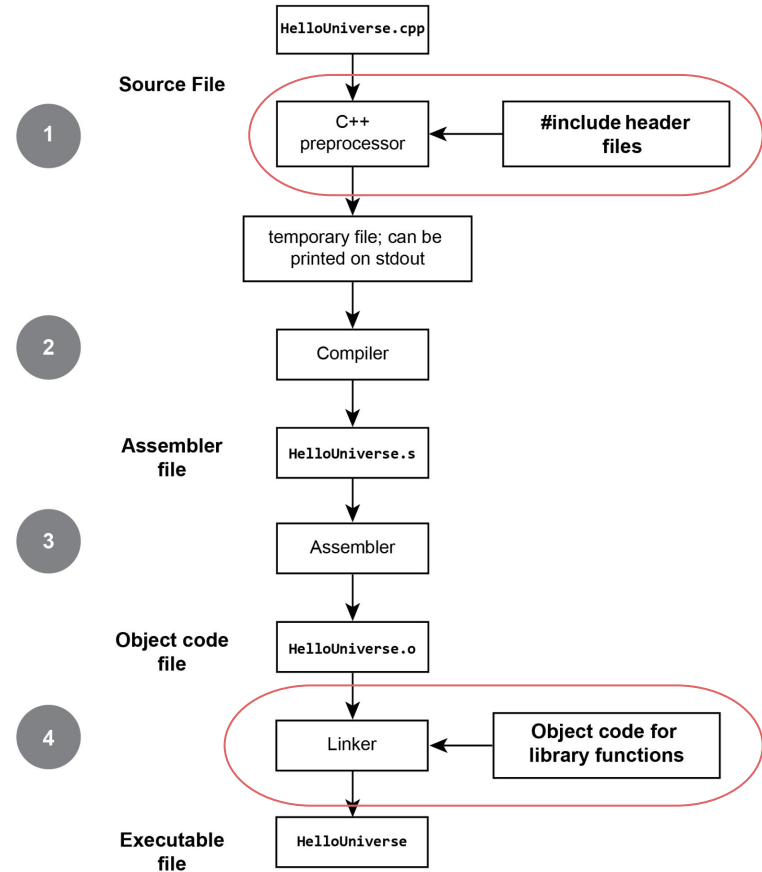
1. Preprocesamiento (.cpp + .h -> .ii)
2. Compilación (.ii -> .o/.a) o (.ii -> .so/.dll)
3. Linking (.o + .a -> .out) o (.so + .dll -> .exe)



Modelo de compilación en C++

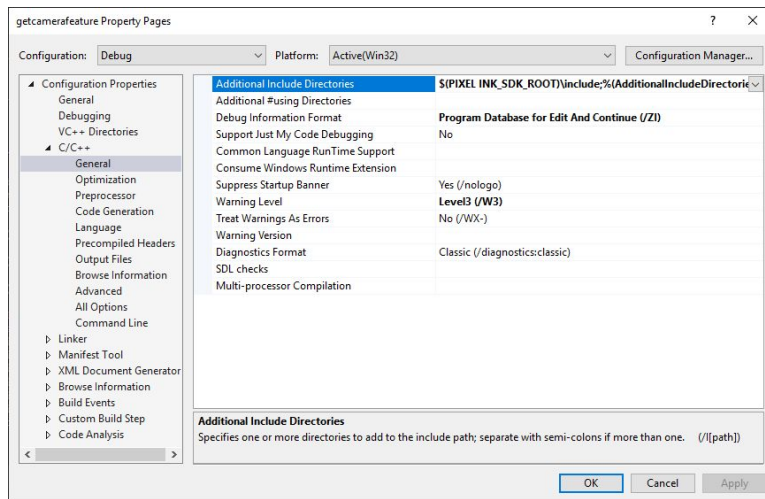
1. Preprocesamiento (.cpp + .h -> .ii)
2. Compilación (.ii -> .o/.a) o (.ii -> .so/.dll)
3. Linking (.o + .a -> .out) o (.so + .dll -> .exe)

Hay que decirle al preprocesador dónde buscar headers, y al linker qué librerías incluir en la compilación y también dónde buscarlas.



¿Cómo le decimos al preprocesador dónde buscar archivos?

En Visual Studio: Además de la librería estándar, se configura la variable “Additional Include Directories”



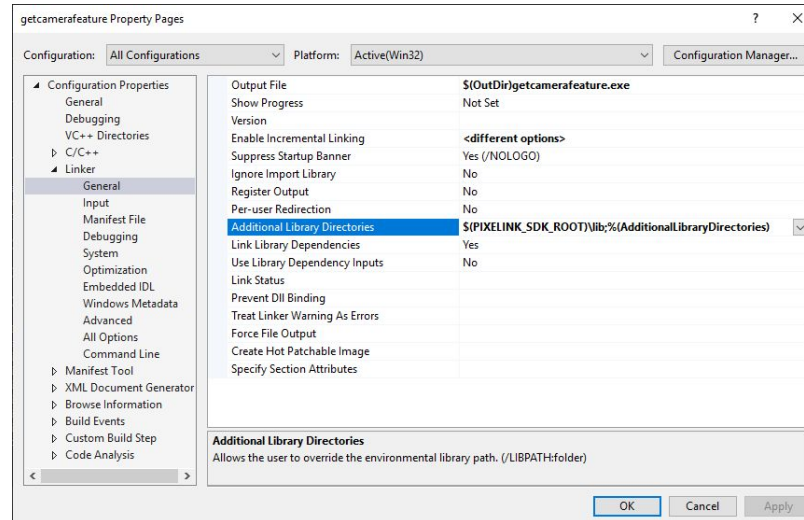
¿Cómo le decimos al preprocesador dónde buscar archivos?

Compilando con gcc: Se añade la opción guion i mayúscula.

Ejemplo: gcc -Ipath/to/dir/with/headers archivo.cpp

¿Cómo le decimos al linker dónde buscar archivos?

En Visual Studio: Además de la librería estándar, se configura la variable “Additional Library Directories”



¿Cómo le decimos al linker qué librerías buscar?

Compilando con gcc: Se añade la opción guion l minúscula

Ejemplo: `gcc -lglut -lpthead archivo.cpp`

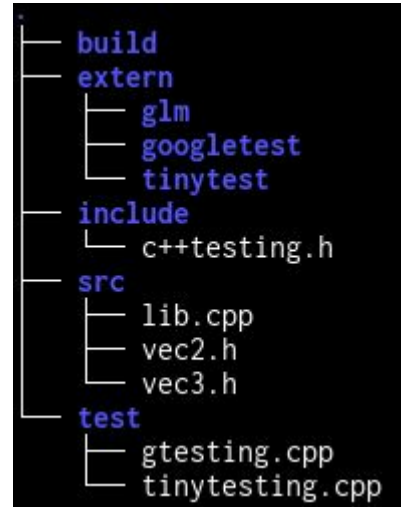
¿Cómo le decimos al linker donde buscar librerías?

Compilando con gcc: Se añade la opción guion l
mayúscula

Ejemplo: gcc -Lpath/to/dir/with/libraries archivo.cpp

Estructura típica de una librería en C++

- extern: Carpeta con dependencias externas.
- include: Carpeta con headers que otras librerías van a usar (típicamente usados también para testear).
- src: Código de la librería (implementación de las funciones declaradas en include).
- test: test.
- build: Carpeta donde se dejan los archivos compilados y ejecutables.

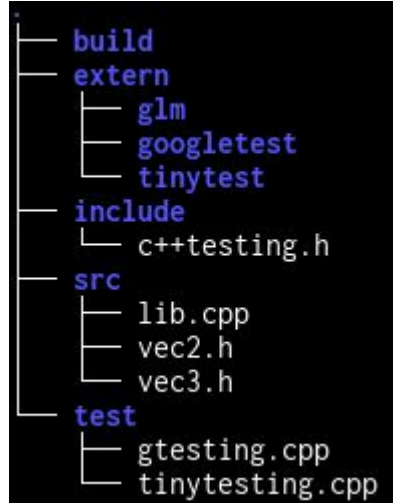


¿Cómo se compilan los test entonces?

```
.
├── build
├── extern
│   ├── glm
│   ├── googletest
│   └── tinytest
├── include
│   └── c++testing.h
├── src
│   ├── lib.cpp
│   ├── vec2.h
│   └── vec3.h
└── test
    ├── gtesting.cpp
    └── tinytesting.cpp
```

¿Cómo se compilan los test entonces?

- Cosas que compilar:
 - Se compilan las librerías externas (glm, googletest y tinystest), y se dejan en build.
 - Se compila build.cpp y se deja la librería en build.
- Preprocesador:
 - Se añaden a la variable del preprocesador los headers de las librerías, y la carpeta include.
- Linker:
 - Se añaden a la variable del linker las carpetas en donde se guardan las librerías.



¿Cómo automatizar este proceso?

Problemas:

- Determinar qué es necesario compilar de nuevo cuando se editar una librería o un ejecutable.
- Compilar código en distintos SOs o en distintos compiladores.
- Testear o hacer profiling del proceso de compilación.
- Problemas de escala.

Automatizando el proceso: Build systems

Problemas que arreglan:

- Determinar qué es necesario compilar de nuevo cuando se editar una librería o un ejecutable.
- Compilar código en distintos SOs o en distintos compiladores.
- Testear o hacer profiling del proceso de compilación.
- Problemas de escala.

Build Systems

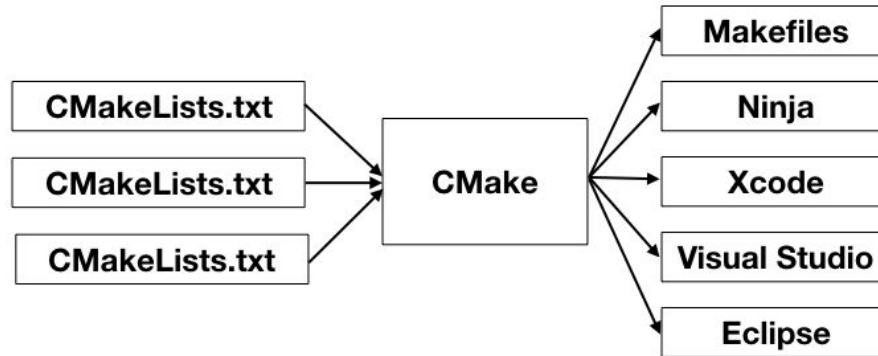
- [Bazel](#) – A multi-language, fast and scalable build system from Google. [Apache]
- [Bear](#) – A tool to generate compilation database for clang tooling. [GPLV3]
- [Buck](#) – A fast build system that encourages the creation of small, reusable modules over a variety of platforms and languages including C++ developed and used at Facebook. Written in Java. [Apache]
- [clib](#) – Package manager for the C programming language. [MIT]
- [CMake](#) – Cross-platform free and open-source software for managing the build process of software using a compiler-independent method. [BSD]
- [C++ Archive Network](#) – Cross-platform C++ Dependency Manager with a lot of packages available.
- [Cget](#) – Cmake package retrieval. [Boost] [website](#)
- [Conan](#) – C/C++ Package Manager, open sourced. [MIT]
- [CPM](#) – A C++ Package Manager based on CMake and Git.
- [FASTBuild](#) – High performance, open-source build system supporting highly scalable compilation, caching and network distribution.
- [Hunter](#) – CMake driven cross-platform package manager for C++. [BSD-2]
- [MesonBuild](#) – An open source build system meant to be both extremely fast, and, even more importantly, as user friendly as possible.
- [Ninja](#) – A small build system with a focus on speed.
- [Scons](#) – A software construction tool configured with a Python script.
- [Sconsolidator](#) – Scons build system integration for Eclipse CDT.
- [Spack](#) – A flexible package manager that supports multiple versions, configurations, platforms, and compilers. [Apache-2.0/MIT]
- [tundra](#) – High-performance code build system designed to give the best possible incremental build times even for very large software projects.
- [tup](#) – File-based build system that monitors in the background for changed files.
- [Premake](#) – A tool configured with a Lua script to generate project files for Visual Studio, GNU Make, Xcode, Code::Blocks, and more across Windows, Mac OS X, and Linux.
- [Vcpkg](#) – C++ library manager for Windows, Linux, and MacOS. [MIT]
- [waf](#) – Python-based framework for configuring, compiling and installing applications. [BSD] [website](#)
- [XMake](#) – A C/C++ cross-platform build utility based on Lua. [Apache]

¿Por qué CMake?

- Cross Platform: Funciona en Windows, Linux, Mac, etc.
- +Antiguo: Se usa todavía en mucho código antiguo
- Usado en muchos proyectos de código libre
- Soportado por varios IDEs: CLion, Visual Studio, etc.
- Compiler independent: Visual C++, gcc, Clang, etc.

¿Por qué CMake?

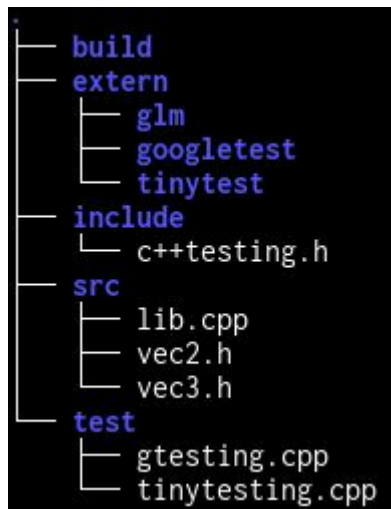
- Cross Platform: Funciona en Windows, Linux, Mac, etc.
- +Antiguo: Se usa todavía en mucho código antiguo
- Usado en muchos proyectos de código libre
- Soportado por varios IDEs: CLion, Visual Studio, etc.
- Compiler independent: Visual C++, gcc, Clang, etc.



¿CMake Moderno?

- Versiones de CMake 3.4+
- Más rápido y fácil de escribir que versiones anteriores

Estructura típica de una librería en C++/CMake



Modelo CMake moderno

targets:

- executable -> .out
- libraries -> .a .o

Los targets se configuran usando las funciones:

- target_include_directories -> Añade carpetas al path del preprocesador de un target.
- target_link_libraries -> Añade libraries (que son targets) al linker de otro target.

Hay más funciones para configurar targets*

```
.
├── CMakeLists.txt
├── extern
│   ├── glm
│   ├── googletest
│   └── tinytest
├── include
│   └── c++testing.h
├── src
│   ├── CMakeLists.txt
│   ├── lib.cpp
│   ├── vec2.h
│   └── vec3.h
└── test
    ├── CMakeLists.txt
    ├── gtesting.cpp
    └── tinytesting.cpp
```

Modelo CMake moderno

Ejemplo:

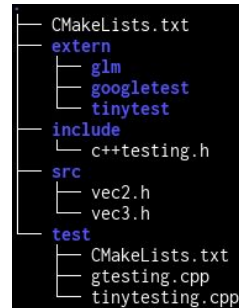
```
# test/CMakeLists.txt

set(LIBS
    ${LIBS}
    glm
    gtest
)

set(INCLUDES
    ${INCLUDES}
    ../extern/glm
    ../extern/googletest/include
    ../extern/tinytest
    ../include/
)

add_executable(tinytesting tinytesting.cpp)
target_link_libraries(tinytesting PRIVATE ${LIBS})
target_include_directories(tinytesting PRIVATE ${INCLUDES})

add_executable(gtesting gtesting.cpp)
target_link_libraries(gtesting PRIVATE ${LIBS})
target_include_directories(gtesting PRIVATE ${INCLUDES})
```



Ejemplo de código en tiempo real

