Compiling a single source file into an

Executable

Bu bölümde CmakeLists.txt dosyasını nasıl yapılandırağımızı ve tek bir source dosyadan tek bir executable dosyanın nasıl oluşturulacağını göreceğiz. Bu bölümde C++ diline değineceğiz ancak C ve fortran içinde cmake kullanılabilir.

Şimdi aşağıdaki kodu tek executable dosyasına compile edelim.

#include <iostream>

void say\_hello(){ return std::string{"Hello World"};}

int main(){

    std::cout << say\_hello() << std::endl;

    return -1;

}

Nasıl Yapacağız?

Şimdi main.cpp’nin yanında bir CmakeLists.txt dosyası oluşturmamız gerekecek. Bu derleyinin hangi dosyaları nasıl derleyeceği veya hangi build generator ün kullanıcağını seçicez.

1. CmakeLists.txt adında bir dosya oluşturalım ve dosyayı herhangi bir text editor ile açalım

2. ilk satıra cmake minumum gereklilik satırını yazalım ve daha aşağı bir cmake sürümü görürsek fatal error verdirelim.

cmake\_minimum\_required(VERSION 3.15 FATAL\_ERROR)

3. CmakeLists.txt dosyasında ikinci satıra projenin adı ve projenin hangi dilde olduğunu belirtiyoruz.

project(recipe-01 LANGUAGES CXX )

4. Cmake yeni bir target gösteriyoruz ve bu targeta source file’in hem compile edilmesini hemde link edilmesini sağlıyoruz.

add\_executable(main main.cpp)

5. CmakeLists.txt dosyasını kayıt ediyoruz.

6. Şimdi Command line açarak aşağıdaki komutları giriyoruz.

$ mkdir -p build

$ cd build

$ cmake ..

Yukarıdaki kod çalıştırdıktan sonra aşağıdaki çıktıyı almamız gerekiyor

-- Building for: Visual Studio 17 2022

-- The CXX compiler identification is MSVC 19.34.31933.0

-- Detecting CXX compiler ABI info

-- Detecting CXX compiler ABI info - done

-- Check for working CXX compiler: C:/Program Files/Microsoft Visual Studio/2022/Professional/VC/Tools/MSVC/14.34.31933/bin/Hostx64/x64/cl.exe - skipped

-- Detecting CXX compile features

-- Detecting CXX compile features - done

-- Configuring done

-- Generating done

-- Build files have been written to: C:/Users/CMLDMR/Desktop/CMAKE\_TUT/build/build

Herşey normal gittiğine göre artık kodumuzu derleyebiliriz.

$ cmake --build .

Nasıl Çalışıyor?

Bu bölümde basit bir CmakeLists.txt dosyasını build etmek için kullandık.

cmake\_minimum\_required(VERSION 3.15 FATAL\_ERROR)

project(recipe-01 LANGUAGES CXX )

add\_executable(main main.cpp)

projeyi configure etmek için bir build system oluşturmamız gerekiyor ve bunu da CLI(Command Line Interface) cmake komutunu çalıştırmamız gerekiyor. Cmake CLI bir çok switch bize sunar, bunlara cmake –help ile ulaşabiliriz. Bu kitapta bu switch ile bir çoğuna değinicez. Cmake –help komutu ile ulaştığımız bir çok şeye Cmake Manuel ile ulaşabiliriz. Aşağıdaki komut ile build systemi oluşturabiliriz.

**$ mkdir -p build**

**$ cd build**

**$ cmake ..**

Burada build adında bir klasör oluşturuyoruz. Build system bu klasörün içine oluşturulacak. cd build ile klasörün içine giriyoruz. Cmake .. komutunu çalıştırınca parent klasörün içinde CmakeLists.txt dosyasını okuyorak bulunduğunu klasörün içinde build system ‘in dosyalarını oluşturur.

**$ cmake -H. -Bbuild**

Buraya kadar build klaösür oluşturduk ve içine build system’I oluşturduk. Bu komut cross-platform ve -H ve -B switchlerini içeriyor. -H switch CmakeLists.txt dosyasını CLI nin bulunduğu dizinde arıyor. -Bbuild switch de cmake oluşturduğu bütün dosyaları build klasörünün içine oluşturur.

**$ cmake ..**

Bu komutu çalıştıdığımız da en sonra konfigurasyonun tamamlandığını ve Generating done diye dosyaların oluşturulduğunu görebilirsiniz.

-- The CXX compiler identification is GNU 8.1.0

-- Check for working CXX compiler: /usr/bin/c++

-- Check for working CXX compiler: /usr/bin/c++ -- works

-- Detecting CXX compiler ABI info

-- Detecting CXX compiler ABI info - done

-- Detecting CXX compile features

-- Detecting CXX compile features - done

-- Configuring done

-- Generating done

-- Build files have been written to: /home/user/cmake-cookbook/chapter-01/recipe-01/cxx

Cmake bir build system generatorudur. Peki build system nedir? Mesela Unix MakeFiles, Ninja, Visual Studio gibi… build generatorlar sizin kaynak dosyalarını derlemenize yardımcı olur. Cmake seçtiğiniz build system generator göre instruction dosyaları oluşturur. Cmake Unix Makefiles generator kullanır. Windows platformunda varsayılan generator visual studio dur. Generator lara bir sonraki bölümde daha yakından bakıcaz ve daha ileriki bölümde tekrardan gözden geçirip alternatif generatorlara bakıcaz.

GNU/Linux platformunda varsayılan generatır Unix MakeFiles dır. Unix MakeFiles aşağıdakiler ile build eder.

* *Makefile* : *make* komutunu bir çok komut seti ile çalıştırıp projeninizi build eder.
* *CmakeFiles*: bu klasör geçici bir klasördür ve işletim sistemini, derleyiciyi gibi belirler. Generator’a bağlı olarakta projenin özel dosyalarını da içerir.
* *Cmake\_install.cmake:* bu dosya *install* kurallarını içerir. Bunu projenizi install ederken kullanacağız.
* *CmakeCache.txt:* adından anlaşılacağı gibi cmake configurasyon yeniden çağırdığımız da kullanılır.

Aşağıdaki komutu kullandığımız da projemizi build etmiş oluyoruz.

**$ cmake --build .**

Bu komut generic bir komuttur ve cross-platform dur. Kullandığımız platforma göre build systemini çağırır.

Son Olarak ta Cmake özel bir ada yada özel bir dizini sizden zorunluluk olarak istemez. Fakat siz bunu belirttiğiniz zaman özel bir ada yada klasöre build eder.

**$ mkdir -p /tmp/someplace**

**$ cd /tmp/someplace**

**$ cmake /path/to/source**

**$ cmake --build .**

**Daha Fazlası**

Resmi döküman sayfasında cmake ile daha fazla detaylar içerir. Cmake tarafından oluşturulan build system, (MakeFiles) object dosyaları, executables ve libraries için target ve kurallar içerir. Hello-word.exe bir targettır ve aşağıdaki seçenekleri vardır.

**$ cmake --build . --target help**

Bu komut çalıştırdığımız da aşağıdaki sonuçları görebiliriz.

... all (the default if no target is provided)

... clean

... depend

... rebuild\_cache

... hello-world

... edit\_cache

... hello-world.o

... hello-world.i

... hello-world.s

Target kendimiz seçmek istersek *cmake –build .. –target <target-name>* komutunu kullanabilir.

* *all :* bu varsayılan bir komuttur ve diğer bütün targetları build eder.
* *clean:* bu komut ile generate ettiğimiz bütün dosyaları temizler.
* *Depend:* eğer bağımlılık varsa cmake bağımlılıkları generate etmek için kullanılır.
* *Rebuild\_cache:* CmakeChache.txt dosyasını yeniden generator eder.
* *Edit\_cache:* cache entry lerini editlemek için kullanılır.

Daha karmaşık projeler için test adımları ve installation rules için cmake ekstra targetları vardır.

* *Test:* test suitiniçağırmak için kullanılır. Ctest kısmını ileriki bölümlerde tartışacağız.
* *Install:*  bu target installation kurallarını içerir. Bunu daha sonra tartışacağız ve bir installer yazacaüız.
* *Package:* bu target Cpack komutunu redistributable package oluşturur. Packaging ve Cpack ileriki bölümlerde tartışacağız.

Switching Generators

Cmake build system generator ve tek CmakeLists.txt dosyası ile farklı platformlar için farklı toolstack leri configure eder. CmakeLists.txt dosyası ile target build system için seçimler yapılabilir. (Unix Makefiles, Ninja, Visual Studio … ). Bu kısmı daha sonra tartışacağız.

**Getting Ready**

Cmake kendi içerisinde bir native build systemleri destekler. Hem CLİ hemde geliştirme ortamı IDE sinin araçlarını destekler.

Cmake in desteklediği generatorları aşağıdaki komut ile ulaşabilirsiniz.

$ cmake --help

Generators

The following generators are available on this platform (\* marks default):

\* Visual Studio 17 2022 = Generates Visual Studio 2022 project files.

Use -A option to specify architecture.

Visual Studio 16 2019 = Generates Visual Studio 2019 project files.

Use -A option to specify architecture.

Visual Studio 15 2017 [arch] = Generates Visual Studio 2017 project files.

Optional [arch] can be "Win64" or "ARM".

Visual Studio 14 2015 [arch] = Generates Visual Studio 2015 project files.

Optional [arch] can be "Win64" or "ARM".

Visual Studio 12 2013 [arch] = Generates Visual Studio 2013 project files.

Optional [arch] can be "Win64" or "ARM".

Visual Studio 11 2012 [arch] = Generates Visual Studio 2012 project files.

Optional [arch] can be "Win64" or "ARM".

Visual Studio 10 2010 [arch] = Deprecated. Generates Visual Studio 2010

project files. Optional [arch] can be

"Win64" or "IA64".

Visual Studio 9 2008 [arch] = Generates Visual Studio 2008 project files.

Optional [arch] can be "Win64" or "IA64".

Borland Makefiles = Generates Borland makefiles.

NMake Makefiles = Generates NMake makefiles.

NMake Makefiles JOM = Generates JOM makefiles.

MSYS Makefiles = Generates MSYS makefiles.

MinGW Makefiles = Generates a make file for use with

mingw32-make.

Green Hills MULTI = Generates Green Hills MULTI files

(experimental, work-in-progress).

Unix Makefiles = Generates standard UNIX makefiles.

Ninja = Generates build.ninja files.

Ninja Multi-Config = Generates build-<Config>.ninja files.

Watcom WMake = Generates Watcom WMake makefiles.

CodeBlocks - MinGW Makefiles = Generates CodeBlocks project files.

CodeBlocks - NMake Makefiles = Generates CodeBlocks project files.

CodeBlocks - NMake Makefiles JOM

= Generates CodeBlocks project files.

CodeBlocks - Ninja = Generates CodeBlocks project files.

CodeBlocks - Unix Makefiles = Generates CodeBlocks project files.

CodeLite - MinGW Makefiles = Generates CodeLite project files.

CodeLite - NMake Makefiles = Generates CodeLite project files.

CodeLite - Ninja = Generates CodeLite project files.

CodeLite - Unix Makefiles = Generates CodeLite project files.

Eclipse CDT4 - NMake Makefiles

= Generates Eclipse CDT 4.0 project files.

Eclipse CDT4 - MinGW Makefiles

= Generates Eclipse CDT 4.0 project files.

Eclipse CDT4 - Ninja = Generates Eclipse CDT 4.0 project files.

Eclipse CDT4 - Unix Makefiles= Generates Eclipse CDT 4.0 project files.

Kate - MinGW Makefiles = Generates Kate project files.

Kate - NMake Makefiles = Generates Kate project files.

Kate - Ninja = Generates Kate project files.

Kate - Unix Makefiles = Generates Kate project files.

Sublime Text 2 - MinGW Makefiles

= Generates Sublime Text 2 project files.

Sublime Text 2 - NMake Makefiles

= Generates Sublime Text 2 project files.

Sublime Text 2 - Ninja = Generates Sublime Text 2 project files.

Sublime Text 2 - Unix Makefiles

= Generates Sublime Text 2 project files.

Bu bölümde bu generatorlar arasında aynı proje üzerinde nasıl rahatlık geçiş yapabildiğimizi göstereceğiz.

Bir önceki bölümde kullanığımız örneği bu bölümde builder farklı seçerek tekrar kullanalım.

1. Proje ilk önce aşağıdaki komuta göre configure edelim.

**$ mkdir -p build**

**$ cd build**

**$ cmake -G Ninja ..**

-- The C compiler identification is GNU 11.2.0

-- The CXX compiler identification is GNU 11.2.0

-- Detecting C compiler ABI info

-- Detecting C compiler ABI info - done

-- Check for working C compiler: C:/Qt/Tools/mingw1120\_64/bin/gcc.exe - skipped

-- Detecting C compile features

-- Detecting C compile features - done

-- Detecting CXX compiler ABI info

-- Detecting CXX compiler ABI info - done

-- Check for working CXX compiler: C:/Qt/Tools/mingw1120\_64/bin/c++.exe - skipped

-- Detecting CXX compile features

-- Detecting CXX compile features - done

-- Configuring done

-- Generating done

-- Build files have been written to: C:/Users/CMLDMR/Desktop/CMAKE\_TUT/build

2. ikinci olarak ta build komutunu çalıştıralım.

$ cmake --build .

[1/2] Building CXX object CMakeFiles/helloWorld.dir/main.cpp.obj

[2/2] Linking CXX executable helloWorld.exe

Nasıl Çalışıyor?

Bir önceki bölüme göre configuration adımlarında herhangi bir değişiklik yapmadık. Fakat compile adımında ve build dizinin içerisi bir öncekinden farklı çünkü her generator kendi özel dosyaları bulunmakta;

* *Build.ninja* ve *rules.ninja:* buind adımlarını ve ninja için build kurallarını içeriyor.
* *CmakeCache.txt:* generatore göre cache dosyalarını içeriyor.
* *CmakeFiles:* configuration sırasında oluşturulan geçici dosyaları içerir.
* *Cmake\_install.cmake:* install zamanında install kurallarını içerir.

Şimdi *cmake –build .*  komutunu *ninja* builder ile çalıştırarak cross-platform derleme işlemini başlatıyoruz.

Building and linking static and

shared libraries

Projeler çoğu zaman birden fazla yürütülebilir dosyalar içerir. Projeler birden fazla kaynağa bölünürler, mevcut dizin altına alt dizin olarak source tree olarak dağıtılırlar. Bu bölümde sadece source code organize etmiyeceğiz, modularity, code reuse, ayrı ilgili kısımlar, bir çok ortak taskları tek bir library de birleştirmeyi de sağlıyacapız. Bu yöntem bize kodu basitliğe indirgemeyi ve projeyi geliştirirken compile hızınıda artıracağız. Bu bölümde kaynakları bir library e nasıl gruplayacağımızı ve bunu projemize nasıl linkleyeceğimizi anlatacağız.

**Getting Ready**

Bir önceki projemize bakalım. Ancak tek kaynak dosya yerine, bir message print eden bir class wrapper I oluşturacağız.

* Ilk olarak daha önce message.h ve message.cpp adında iki dosya oluşturalım ve içlerini aşağıdaki gibi dolduralım ve main.cpp dosyamızı değiştirelim.

message.h

#pragma once

#include <iosfwd>

#include <string>

class Message{

    public:

    Message( const std::string &msg):mMessage(msg){}

    friend std::ostream &operator<<(std::ostream &os , Message &obj){

        return obj.print(os);

    }

    private:

    std::string mMessage;

    std::ostream &print(std::ostream &os);

};

Message.cpp

#include "message.h"

#include <iostream>

#include <string>

std::ostream &Message::print( std::ostream &obj ){

    obj << mMessage;

    return obj;

}

Main.cpp

#include <iostream>

#include "message.h"

int main(){

    Message msg("Hello World From Static Library");

    std::cout << msg << std::endl;

    return -1;

}

* Şimdi CmakeLists.txt dosyasına aşağıdaki kod satırlarını sırasıyla ekleyelim
* add\_library(message
* STATIC
* message.h
* message.cpp)

bu kod parçası message adında bir class olarak derleyerek message adında bir static library oluşturacak.

target\_link\_libraries(helloWorld message)

bu kod satırı ile de message adında bir static library yi projemize link leyecek ve Message sınıfını kullanabileceğiz.

$ cmake --build .

Consolidate compiler generated dependencies of target message

[ 25%] Building CXX object CMakeFiles/message.dir/message.cpp.obj

[ 50%] Linking CXX static library libmessage.a

[ 50%] Built target message

[ 75%] Building CXX object CMakeFiles/helloWorld.dir/main.cpp.obj

[100%] Linking CXX executable helloWorld.exe

[100%] Built target helloWorld

Cmake –build komutunu tekrar çağırdığımız da gördüğünüz gibi ilk önce (libmessage.a)static library oluşturacak ve oluşturduğu bu library ana projeye bağlayacak.

$ ./helloWorld.exe

Hello World From Static Library

Yukardaki gibi farklı bir yerden ekranımıza yine print yaptık.

**How it Works**

Daha önce ki bölümlerden farklı olarak 2 yeni keyword’umuz var.

* *add\_library(message STATIS message.h message.cpp):* bu komut satırı \*.h ve \*.cpp dosylarını bir static library eklemek için komut setlerini oluşturur. *Add\_library* ilk parametresi target adı yani **message.** Bu ismi CmakeLists.txt içerisinde referans alarak kullanabileceğiz. Cmake library dosyasını oluştururken bu ismin başında **lib** ekleyerek dosya adını oluşturur(**libmessage.a**). daha sonra gelen **STATIC**  kelimeside library tipinin nasıl olucağına ve buna göre de library dosyasının uzantısının ne olucağına karar verir. Bu genellikle işletim sistemine göre değişir.
* *target\_link\_libraries(helloWorld message):* bu kod satırıda message adında oluşturduğumuz library ana projemizin executable dosyasına link eder. Bu kod executable dosyamızın artık message library olmadan çalışmayacağını garanti eder. Message library nin ana proje executable dosyasından önce build edildiğinden emin olun.

derleme tamamlandıktan sonra, build dizini içerisinde libmessage.a adında yeni bir dosya göreceğiz. Bu dosya GNU/Linux ortamında \*.a uzantısında bir static library dosyasıdır.

* *STATIC:* bu kelime bahsettiğimiz gibi static library yaratırken derleyicinin oluşturduğu object dosyalarını arşivleyerek ana programda yürütülebilir olarak link eder.
* *SHARED:* bu kelime shared bir library oluşturur ve static library deki gibi objleri arşivlemez. Oluşturulan objeleri tekrar derler ve dynamic link library diye tabir ettiğimiz bir \*.dll yada \*.so dosyası oluşturur. Ayrıca bir de \*.lib dosyası oluşturur. Derleme anında \*.lib dosyası kullanılır ancak linkleme yapılmaz. Runtime anında lib dosyanında içinde bahsedilen \*.dll yada \*.so dısyasını arar ve entry kısımlarında giriş yaparak kdu yürütür.
* *OBJECT:* bu kelimde daha ne static nede shared library deki gibi object leri oluşturduktan sonra linkleme yapmaz. Bunu daha sonra static yada shared olarak tekrar derlemek için kullanılabilir.
* *MODULE:* bu kelime shared gibi bir library oluşturur ve linkler ancak ana projemiz bunu ihtiyaç duyduğunda runtime anında yüklemesini yapar ve kullanır. Bu library kullanana kadar ana projenin çalışması için bir ihtiyaç zorunluluğu gerektirmez.

Cmake ayrıca başka özel kütüphaneler de generate edebilir. Bu library ler herhangi bir çıktı vermezler, shared, static yada diğerleri gibi, ancak kodun bağımlılıklarını ve build gerekliliklerini arasında çok faydalı bir organisation yapar.

* *IMPORTED:* bu library tipi, library nin projenin dışında bir yerde oluduğunu ifader eder. Bu library nin ana kullanım amacı upstream package ile daha önceden var olan bir library nin kullanıldığını ifade eden değişmeyen librarylerdir. Bunu kitabın geri kalanında tekrar ele alacağız.
* *INTERFACE:* bu library Cmake özel bir library dir ve IMPORTED library ye benzer ancak değişebilir bir librarydir. Conda Package leri gibi projenin bağımlılıkları ile dağıtımında kullanılır.
* *ALIAS:* -----

Bu örnekte *add\_library* kullanarak doğrudan library leri oluşturulmalarını gördük. Daha sonraki bölümlerde *target\_sources* cmake komutlarının nasıl kullanılacağını, özellikle ana projemizin nasıl yapılandıracağımızı göreceğiz.

**There is More**

Şimdi projemizde daha fazla neler var baştan CmakeLists.txt dosyasını tekrar kontrol edelim.

cmake\_minimum\_required(VERSION 3.15 FATAL\_ERROR)

project(helloWorld LANGUAGES CXX)

add\_library(message-objs

            OBJECT

            message.h

            message.cpp

            )

add\_library(message-shared

            SHARED

            $<TARGET\_OBJECTS:message-objs>)

add\_library( message-static

            STATIC

            $<TARGET\_OBJECTS:message-objs>

            )

add\_executable(helloWorld main.cpp)

target\_link\_libraries(helloWorld message-shared)

CmakeLists.txt dosyasını yukardaki gibi değiştirelim. Yukardaki görüldüğü gibi ilk önce OBJECT library oluşturan bir *add\_lirary* dosyası bulunmaktadır. Bu oluşturulan libraryden artık hem *static* hemde *shared* library oluşturulabilir. Dikkat edilmesi gereken yer *$<TARGET\_OBJECT:message-objs>* kısmı. Static ve shared library ler bu object library lere bağlanıyor.

Peki Bu OBJECT library leri her zaman link edebilirmiyiz? Bu işletim sistemine ve derleyiciye bağladır.

1. GNU/Linux ve MacOS işletim sistemlerinde seçilen derleyiciye göre değişiklik gösterir.

2. Windows İşletim sisteminde sadece MinGW vs MSYS2 derleyicisinde çalışır. Visual Studio da böyle bişey yoktur.

Controlling compilation with

Conditionals

Şimdiye kadar cmake projelerinde kısmi olarak projelendirmeye baktık. Cmake in lineer yürütme akışında; source dosyalarından tekli executable dosyaya kadar, shared yada static olur. Cmake execution flow’da, configuration da, complitation ve linkage de kendi dilini kullanır. Bu bölümde şartlı yapılandırmayı göreceğiz.

**How to do it?**

Şimdi son yaptığımız proje üzerinden devam edelim. Iki davranız arasında seçim yapmayı sağlayalım.

* Message.h ve message.cpp dosyasını library e atarkeni static yada shared olma durumunu seçip seçtiğimiz duruma göre helloWorld.exe ye link edelim.
* Message.h ve message.cpp dosyasını build edip tek bir executable dosyasına bir library üretmeden oluşturalım.

CmakeLists.txt dosyasını yeniden yapılandıralım:

1. set(USE\_LIBRARY OFF) adında bir kod satırı ekleyelim.

set( USE\_LIBRARY OFF )

message(STATUS “Compile source into library?” ${USE\_LIBRARY})

* shared library oluşturacaz mı oluşturmuycaz mı onun kararını veren bir kod satırı daha ekleyelim

set( BUILD\_SHARED\_LIBS OFF )

* \_source adında bir değişkene source dosyalarını append edelim.

list( APPEND \_source message.h message.cpp )

* şimdi library oluşturacaksak bir if else kontrollürü ile iki farklı bir executable dosyası oluşturma yöntemi belirleyecez.

cmake\_minimum\_required(VERSION 3.15 FATAL\_ERROR)

project(helloWorld LANGUAGES CXX)

set( USE\_LIBRARY OFF )

set( BUILD\_SHARED\_LIBS OFF )

list( APPEND \_source message.h message.cpp )

if( USE\_LIBRARY )

    # add library will create shared libs and link

# BUILD\_SHARED\_LIBS OFF

    add\_library( message-shared ${\_source} )

    add\_executable( helloWorld main.cpp )

    target\_link\_libraries(helloWorld message-shared )

else()

    add\_executable(helloWord main.cpp ${\_source})

endif()

* Başlangıçta USE\_LIBRARY OFF yaptığımız için herhangi bir library oluşturmadan Message Sınıfını direk executable içine gömecek.

**How it Works?**

Iki tane yeni değişkene giriş yaptık. Bunlar USE\_LIBRARY ve BUILD\_SHARED\_LIB. Her ikisinide OFF yaptık. Cmake dökümanında true yada false çeşitli yöntemlerle gösterilebilir.

* True gösterimi: 1 , true , ON , YES, Y yada 0 olmayan herhangi bir sayı ile gösterilebilir.
* False gösterimi: 0, false, OFF, NO,IGNORE, NOTFOUND, N, boş bir string yada -NOTFOUND son eki ile biten bir değişken ile gösterilebilir.

USE\_LIBRARY iki davranış arasında seçim yapar. BUILD\_SHARED\_LIBS Cmake için global bir değişkendir. Hatırlayacağımız üzere add\_library kodu ile static/shared/object olarak library ler oluşturabiliyorduk. BUILD\_SHARED\_LIBS global bir değişken olduğundan herhangi bir tanımı olmadığında veya bu değişken false yapıldığında cmake bunu static olarak kabul eder ve add\_library static olarak linklenir.

Bu örnekte cmake akışı içerisinde bir şartlı akışında nasıl oluştuğunu gördük. Ancak bu USE\_LIBRARY kısmını CmakeLists.txt dosyasını texteditor ile değiştirmeden dışardan toggle edemiyoruz. Biz bunun kullanıcı tarafından kodların değiştirilmeden müdahele edilmesini istiyoruz.

Şimdi kullanıcıya bunun ON mu OFF olduğunu soran bir kod satırı ekleyecez. bunu “***option”*** kod satırı ile yapacağız. Set(USE\_LIBRARY OFF) kısmını aşağıdaki kod ile değiştirelim.

option( USE\_LIBRARY "Compile without Shared Library" OFF )

bu kod satırını kullanarak kullanıcı USE\_LIBRARY değişkeninin değerini değiştirebilir ve CmakeListst.txt dosyasına bunun müdahe etmeden yapar. Herhangi bir müdahele yapılmadığı zaman değer örnekte ki gibi OFF olarak kabul edilir.

$ cmake .. -D USE\_LIBRARY=ON

-- Configuring done

-- Generating done

-- Build files have been written to: C:/Users/CMLDMR/Desktop/CMAKE\_TUT/build

Yukardaki configure komutu ile dışardan USE\_LIBRARY değerini ON yaptık ve CmakeLists.txt dosyasına müdahele etmedik. -D switch parametresi ile CmakeLists.txt içerisinde herhangi bir değişkeni set edebiliriz.

**How it Works?**

*Option* kodu 3 parametre ile çağırılır;

* Option değişkeninin adı
* “option ne olduğu ile alakalı bir açıklama”
* Varsayılan bir değer

Option(USE\_LIBRARY “library oluşturulup linklensin mi?” OFF) gibi.

**There is more**

Bazen option içindeki değerler başka bir option içerisinde ki değerlere bağlı olur. Bunun için CmakeLists.txt dosyası için cmake\_dependent\_option() kodu bir option diğer bir optiona bağlaı olduğunu kullanabilmek içindir.

include(CMakeDependentOption)

cmake\_dependent\_option(

    MAKE\_STATIC\_LIBRARY "Static Library" OFF

    "USE\_LIBRARY" ON

)

cmake\_dependent\_option(

    MAKE\_STATIC\_LIBRARY "Static Library" ON

    "USE\_LIBRARY" ON

)

Yukardaki kod Satırını Projemize eklediğimiz de eğer USE\_LIBRARY ON olursa MAKE\_STATIC\_LIBRARY varsayılan olarak OFF gelecek ancak bir bunu CLI ON yapacağız.

$ cmake .. -D USE\_LIBRARY=ON -D MAKE\_STATIC\_LIBRARY=ON

USE\_LIBRARY ON olursa MAKE\_STATIC\_LIBRARY varsayılan olarak OFF gelir ancak CLI ile biz bunu ON yaptık.

Daha önce bahsettiğimiz gibi Cmake mekanısmasında cmake kendi syntaxları yerine başka kullanım tipleride mevcuttur. Bu örnekte, cmake\_dependent\_option() ile if else kullanmadan iki davranış biçimi sergiledik.

Specifying the compiler

Şimdiye kadar derleyici seçimi ile alakalı herhangi bir konuya değinmedik. Cmake platform ve generator seçimi için yeterli seviyede uygun bir araçtır. Cmake ayrıca compiler flaglerini mantıklı bir şekilde kümeleyebilir. Bununla birlikte bu bölümde compiler seçiminin nasıl yapılacağını göreceğiz. Son bölümde build type ve compiler flag nasıl seçiceğimizi göreceğiz.

**How to do it**

Compiler seçimini nasıl yaparız? Örneğin Intel yada Portlan Group compiler arasında hangisini seçiceğimnize nasıl karar vericez? Cmake her derleyici için *cmake\_<lang>\_compiler* adında bir değişkene sahiptir. *<lang>* desteklenen herhangi bir dildir; cxx, c veya fortran. Kullanıcı bu değişkeni iki yolla seçebilir.

* CLI arayüzünda -D opsiyonu ile

$ cmake -D CMAKE\_CXX\_COMPILER=clang++ ..

* Yada environment variable export ederek seçimi yapılabilir; C compiler için cc, Fortran için FC veya C++ için CXX, clang++ derleyicisi için aşağıdaki komutu kullanabiliriz;
* $ env CXX=clang++ cmake ..

Herhangi bir bölümde anlattıklarımız kadarı ile en mantıklı compiler seçimini yapınız.

Biz Cmake kabul ettiği compiler hepsinin yolunu bildiğini varsayarak bu şekilde kod satırını gösteriyoruz aksi halde derleyicinin full path’ini göstermek zorundasınız.

**How it works**

Configure anında, Cmake çalıştığını platform hakkında bir çok testler yaparak proje için kullanılabilir derleyicileri belirler. Uygun derleyici sadece platform için gereklilik değildir, generator içinde bir gerekliliktir. Cmake ilk önce proje dili için compiler belirler. Örneğin eğer C dili ile yazılmış bir projeniz var ise cmake varsayılan olarak cc seçer. GNU/Linux için Unix Makefiles yada Ninja kullanır ve bunlar için varsayılan derleyice C ve C++ için GCC dir. Windows için ise Visual Studio seçilebilir. MinGW derleyicisi için ise MinGW yada MSYS Makefiles generator seçilebilir.

**There is more**

Bizim mevcutta çalıştığımız platform için en uygun compiler ve compiler flags nasıl belirleyebiliriz? Cmake *–system-information* flag adında bir komut ile sisteminiz hakkında bilgilerinizi ekrana yada bir dosyaya yazdırabilir.

$ cmake --system-information information.txt

Bu komut ile systeminizde mevcutta en uygun derleyiciyi bulur ve information.txt dosyasına yazdırır. Komutu çalıştırdıktan sonra information.txt dosyasını incelediğimiz de CMAKE\_CXX\_COMPILER, CMAKE\_C\_COMPILER ve CMAKE\_Fortran\_COMPILER seçeneklerini ve flaglerini görebilirsiniz.

Cmake compiler için ekstra bilgileride bulunmaktadır.

* *CMAKE\_<LANG>\_COMPILER\_LOADED:* eğer proje için compiler enable edilmiş ise bu değer TRUE dur.
* *CMAKE\_<LANG>\_COMPILER\_ID:* bu compiler string her compiler verdor u için özgündür. Örneğin gcc GNU Compiler, Appleclang MacOS için Clang, msvc visual studio için. Fakat bu değerler her derliyici için var olacak bir şart yok.
* *CMAKE\_COMPILER\_IS\_GNU<LANG>:* bu logic değer, compiler GNU Compiler Collection bir parçası ise TRUE olur. <LANG> kısmı GNU Kelimesini hemen takip eder ve eğer compiler C++ compiler ise CXX, C Compiler ise C, Fortran Compiler ise G77 olarak değer alır.
* *CMAKE\_<LANG>\_COMPILER\_VERSION:* bu değişken, seçilen compiler’ın versiyonunu string olarak tutar. Tutulan version formatı: majon[.minor[.patch[.tweak]]] şeklindedir. Fakat CMAKE\_<LANG>\_COMPILER\_ID değişkeninin içeriği her derleyici için geçerli değildir.

Şimdi bu değişkenleri test edebiliriz.

message(STATUS "CXX Compiler Loaded? ${CMAKE\_CXX\_COMPILER\_LOADED}" )

Bu kodu CmakeLists.txt dosyasına ekledikten sonra aşağıdaki çıktıyı göreceğiz.

-- CXX Compiler Loaded? 1

-- Configuring done

-- Generating done

Switching the build type

Cmake build type ve configuration için kavram vardır ki bunlar *Debug*, *Release* ve toplamda 4 adettir. Aynı anda bunlardan bir tanesi kullanılabilir. Çünkü her biri için ayrı ilgili dosyalar ve propertiesler oluşturulur. Bu build type larını seçebileceğimiz cmake değişkeni, CMAKE\_BUILD\_TYPE değişkenidir. Varsayılan olarak *Debug* tur.

* *Debug:* bu değişken ile Library niz yada executable dosyanız hiç bir optimisazyon yapmaz ve bütün debug symbollerini içerir.
* *Release:* bu değişken library ve executable dosyanız optimisazyon yapar ve herhangi bir debug symbol içermez
* *ReleaseWithDebInfo:* bu değişken ile daha z optimisazyon yapar ve debug symbollerini içerir.
* *MinSizeRel:* bu değişlen ile library yada executable dosyalar için object code size artırmadan optimisazyon yapar.

**How to do it**

Bu bölümde build type nasıl seçileceğine bakacağız.

1. Mininum Cmake versiyonumuzu yazıyoruz ve proje adını ve projede kullanılan dilleride belirtiyoruz.

cmake\_minimum\_required(VERSION 3.15 FATAL\_ERROR)

project(helloWorld LANGUAGES C CXX)

1. Sonra, varsayılan build type seçiyoruz(Bu bölümde Release), kullanıcıya bunu message ile gösteriyoruz. CACHE string değişkenini set ederek cache içeriğini düzenlemiş oluruz.

if( NOT CMAKE\_BUILD\_TYPE )

    set(CMAKE\_BUILD\_TYPE Release CACHE STRING "Build Type" FORCE)

endif()

message( STATUS "Build Type: ${CMAKE\_BUILD\_TYPE}")

1. Daha compile flag lerini message ile kullanıcıya gösterelim.

message( STATUS "C Flag Debug information: ${CMAKE\_C\_FLAGS\_DEBUG}")

message( STATUS "C Flag Release information: ${CMAKE\_C\_FLAGS\_RELEASE}")

message( STATUS "C Flag RelWıthDebInfo information: ${CMAKE\_C\_FLAGS\_RELWITHDEBINFO}")

message( STATUS "C Flag MinSizeRel information: ${CMAKE\_C\_FLAGS\_MINSIZEREL}")

message( STATUS "C++ Flag Debug information: ${CMAKE\_CXX\_FLAGS\_DEBUG}")

message( STATUS "C++ Flag Release information: ${CMAKE\_CXX\_FLAGS\_RELEASE}")

message( STATUS "C++ Flag RelWıthDebInfo information: ${CMAKE\_CXX\_FLAGS\_RELWITHDEBINFO}")

message( STATUS "C++ Flag MinSizeRel information: ${CMAKE\_CXX\_FLAGS\_MINSIZEREL}")

1. Aşağıdaki gibi bir çıktı alıyoruz vederleyiciye hangi parametrelerin gittiğini buradan görebiliriz.

$ cmake ..

-- Build Type: Debug

-- C Flag Debug information: -g

-- C Flag Release information: -O3 -DNDEBUG

-- C Flag RelWıthDebInfo information: -O2 -g -DNDEBUG

-- C Flag MinSizeRel information: -Os -DNDEBUG

-- C++ Flag Debug information: -g

-- C++ Flag Release information: -O3 -DNDEBUG

-- C++ Flag RelWıthDebInfo information: -O2 -g -DNDEBUG

-- C++ Flag MinSizeRel information: -Os -DNDEBUG

-- Configuring done

-- Generating done

**How it works**

Default build type ve CLI arayüzünden nasıl değiştirilebildiğini gördük. Bununla birlikte hem derleyicide optimisazyonların hepsi kapalı olsa hemde Debug bilgileri ON olsada istediğimiz gibi kontrol edebiliriz. Ayrıca derleyiciye bağlı olarak ne tür compiler flag lerin kullanıldığını gördük. Cmake ile çalışırken flag leri açık açık yazdırmak yerine –system-information komutunu CLI de kullanarak çıkan ön ayarların ne olduğunu bulmakta mümkündür. Bir sonraki bölümde farklı derleyiciler ve farklı yapı türleri için compiler flag lerin nasıl kullanılacağını ve ayarlanacağını görecez.

**There is more**

CMAKE\_BUILD\_TYPE build generator için nasıl kullanıldığını ve değişkenlerinin ne olduğunu gösterdik. Çoğu zaman hem Debug hemde Release modda configure etmek faydalıdır, Mesela Derleyici optimisazyon seviyelerinin değerlendirirken. Tek configuration lu generatorlerde, Unix Makefiles, MSYS veya Ninja, projenin tam reconfigure olması için iki kere çalıştırılması gerekiyor. Cmake ayrı çoklu configure generator de destekler. Bu çoğu zaman birden entegre geliştirme ortamında olur; Visual Studio, Xcode gibi aynı anda hem Debug hemde Release modda çalıştırılabilir. Bunu CMAKE\_CONFIGURATION\_TYPES değişkeni ile yapılabilir.

$ cmake .. -G "Visual Studio 17 2022 Win64" -D CMAKE\_CONFIGURATION\_TYPES="Release,Debug"

Bunu şu şekilde de yapabiliriz;

$ cmake --build . --config Release