EBook Gratuito

APPRENDIMENTO cmake

Free unaffiliated eBook created from
Stack Overflow contributors.

#cmake

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: cmake

It is an unofficial and free cmake ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official cmake.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con cmake

Osservazioni

CMake è uno strumento per la definizione e la gestione di build di codice, principalmente per C ++.

CMake è uno strumento multipiattaforma; l'idea è di avere una singola definizione di come è costruito il progetto - che si traduce in definizioni di build specifiche per qualsiasi piattaforma supportata.

Lo fa abbinando a diversi sistemi di costruzione specifici della piattaforma; CMake è un passaggio intermedio che genera input di build per diverse piattaforme specifiche. Su Linux, CMake genera Makefile; su Windows, può generare progetti di Visual Studio e così via.

Il comportamento di generazione è definito nei file `CMakeLists.txt` , uno in ogni directory del codice sorgente. Il file `CMakeLists` ogni directory definisce cosa deve fare il buildsystem in quella specifica directory. Definisce anche quali sottodirectory CMake dovrebbe gestire anche.

Le azioni tipiche includono:

- Costruisci una libreria o un eseguibile da alcuni dei file sorgente in questa directory.
- Aggiungi un percorso file al percorso di inclusione utilizzato durante la compilazione.
- Definire le variabili che il buildsystem utilizzerà in questa directory e nelle sue sottodirectory.
- Genera un file, in base alla configurazione di build specifica.
- Individua una libreria che si trova da qualche parte nell'albero dei sorgenti.

I file finali di `CMakeLists` possono essere molto chiari e chiari, poiché ciascuno ha una portata così limitata. Ognuno gestisce solo la parte della build presente nella directory corrente.

Per le risorse ufficiali su CMake, consultare la documentazione e il tutorial di CMake.

Versioni

Versione	Data di rilascio
3.9	2017/07/18
3.8	2017/04/10
3.7	2016/11/11
3.6	2016/07/07
3.5	2016/03/08
3.4	2015/11/12

Versione	Data di rilascio
3.3	2015/07/23
3.2	2015/03/10
3.1	2014/12/17
3.0	2014/06/10
2.8.12.1	2013/11/08
2.8.12	2013/10/11
2.8.11	2013/05/16
2.8.10.2	2012/11/27
2.8.10.1	2012/11/07
2.8.10	2012-10-31
2.8.9	2012-08-09
2.8.8	2012-04-18
2.8.7	2011-12-30
2.8.6	2011-12-30
2.8.5	2011-07-08
2.8.4	2011-02-16
2.8.3	2010-11-03
2.8.2	2010-06-28
2.8.1	2010-03-17
2.8	2009-11-13
2.6	2008-05-05

Examples

Installazione di CMake

Vai alla pagina di download di CMake e ottieni un file binario per il tuo sistema operativo, ad esempio Windows, Linux o Mac OS X. In Windows fai doppio clic sul file binario da installare. Su

Linux esegui il binario da un terminale.

Su Linux, puoi anche installare i pacchetti dal gestore di pacchetti della distribuzione. Su Ubuntu 16.04 puoi installare la riga di comando e l'applicazione grafica con:

```
sudo apt-get install cmake
sudo apt-get install cmake-gui
```

Su FreeBSD puoi installare la riga di comando e l'applicazione grafica basata su Qt con:

```
pkg install cmake
pkg install cmake-gui
```

Su Mac OSX, se si utilizza uno dei gestori di pacchetti disponibili per installare il software, il più notevole dei quali è MacPorts (MacPorts) e Homebrew (Homebrew), è possibile anche installare CMake tramite uno di essi. Ad esempio, in caso di MacPorts, digitare quanto segue

```
sudo port install cmake
```

installerà CMake, mentre nel caso utilizzi il gestore di pacchetti Homebrew dovrai digitare

```
brew install cmake
```

Dopo aver installato CMake, puoi verificare facilmente procedendo come segue

```
cmake --version
```

Dovresti vedere qualcosa di simile al seguente

```
cmake version 3.5.1

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

Passaggio tra tipi di build, ad es. Debug e release

CMake conosce diversi tipi di build, che di solito influenzano i parametri predefiniti del compilatore e del linker (come le informazioni di debug in fase di creazione) o percorsi di codice alternativi.

Per impostazione predefinita, CMake è in grado di gestire i seguenti tipi di build:

- **Debug** : solitamente una classica build di debug che include informazioni di debug, nessuna ottimizzazione, ecc.
- **Versione** : la tipica versione di rilascio senza informazioni di debug e ottimizzazione completa.
- **RelWithDebInfo::** uguale a *Release* , ma con informazioni di debug.
- **MinSizeRel** : una *versione di rilascio* speciale ottimizzata per le dimensioni.

Le modalità di gestione delle configurazioni dipendono dal generatore che viene utilizzato.

Alcuni generatori (come Visual Studio) supportano più configurazioni. CMake genererà tutte le configurazioni contemporaneamente e potrai selezionare dall'IDE o usare `--config CONFIG` (con `cmake --build`) quale configurazione vuoi costruire. Per questi generatori, CMake farà del suo meglio per generare una struttura di directory di build in modo che i file di configurazioni diverse non si sovrappongano.

I generatori che supportano solo una singola configurazione (come Unix Makefile) funzionano diversamente. Qui la configurazione attualmente attiva è determinata dal valore della variabile CMake `CMAKE_BUILD_TYPE` .

Ad esempio, per scegliere un tipo di build diverso, è possibile eseguire i seguenti comandi da riga di comando:

```
cmake -DCMAKE_BUILD_TYPE=Debug path/to/source
cmake -DCMAKE_BUILD_TYPE=Release path/to/source
```

Uno script CMake dovrebbe evitare di impostare lo stesso `CMAKE_BUILD_TYPE` , poiché generalmente è considerata responsabilità dell'utente farlo.

Per i generatori a configurazione singola, il passaggio alla configurazione richiede la riesecuzione di CMake. È probabile che una build successiva sovrascriva i file oggetto prodotti dalla configurazione precedente.

Semplice progetto "Hello World"

Dato un file di origine C ++ `main.cpp` definisce una funzione `main()` , un file `CMakeLists.txt` accompagnamento (con il seguente contenuto) istruirà *CMake* a generare le istruzioni di compilazione appropriate per il sistema corrente e il compilatore C ++ predefinito.

main.cpp *(C ++ Hello World Example)*

```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
    return 0;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.4)

project(hello_world)

add_executable(app main.cpp)
```

Guardalo dal vivo su Coliru

1. `cmake_minimum_required(VERSION 2.4)` imposta una versione CMake minima necessaria per valutare lo script corrente.

2. `project(hello_world)` avvia un nuovo progetto CMake. Ciò attiverà molta logica interna di CMake, in particolare il rilevamento del compilatore C e C ++ predefinito.

3. Con `add_executable(app main.cpp)` viene creata `app` destinazione build che invoca il compilatore configurato con alcuni flag predefiniti per l'impostazione corrente per compilare `app` eseguibile dal file di origine specificato `main.cpp` .

Riga di comando *(In-Source-Build, not recommended)*

```
> cmake .
...
> cmake --build .
```

`cmake .` esegue il rilevamento del compilatore, valuta il `CMakeLists.txt` nel dato `.` directory e genera l'ambiente di compilazione nella directory di lavoro corrente.

Il `cmake --build .` comando è un'astrazione per la chiamata build / make necessaria.

Riga di comando *(Out-of-Source, raccomandato)*

Per mantenere il tuo codice sorgente pulito da eventuali artefatti di costruzione, dovresti creare build "fuori dall'origine".

```
> mkdir build
> cd build
> cmake ..
> cmake --build .
```

Oppure CMake può anche astrarre i comandi di base della tua piattaforma shell dall'esempio sopra riportato:

```
> cmake -E make_directory build
> cmake -E chdir build cmake ..
> cmake --build build
```

"Hello World" con più file sorgente

Per prima cosa possiamo specificare le directory dei file di intestazione da `include_directories()` , quindi dobbiamo specificare i corrispondenti file di origine dell'eseguibile di destinazione da `add_executable()` , e accertarci che esista esattamente una funzione `main()` nei file di origine.

Di seguito è riportato un semplice esempio, tutti i file vengono presupposti inseriti nella directory `PROJECT_SOURCE_DIR` .

main.cpp

```
#include "foo.h"

int main()
{
```

```
    foo();
    return 0;
}
```

foo.h

```
void foo();
```

foo.cpp

```
#include <iostream>
#include "foo.h"

void foo()
{
    std::cout << "Hello World!\n";
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.4)

project(hello_world)

include_directories(${PROJECT_SOURCE_DIR})
add_executable(app main.cpp foo.cpp)  # be sure there's exactly one main() function in the
source files
```

Possiamo seguire la stessa procedura nell'esempio precedente per costruire il nostro progetto. Quindi l' `app` esecuzione verrà stampata

```
>./app
Hello World!
```

"Hello World" come libreria

Questo esempio mostra come distribuire il programma "Hello World" come libreria e come collegarlo ad altri target.

Supponiamo di avere lo stesso set di file sorgente / header come nell'esempio http://www.Scriptutorial.com/cmake/example/22391/-hello-world--with-multiple-source-files . Invece di costruire da più file sorgente, possiamo prima distribuire `foo.cpp` come libreria usando `add_library()` e successivamente collegandolo con il programma principale con `target_link_libraries()` .

Modifichiamo **CMakeLists.txt** in

```
cmake_minimum_required(VERSION 2.4)

project(hello_world)
```

```
include_directories(${PROJECT_SOURCE_DIR})
add_library(applib foo.cpp)
add_executable(app main.cpp)
target_link_libraries(app applib)
```

e seguendo gli stessi passi, otterremo lo stesso risultato.

Leggi Iniziare con cmake online: https://riptutorial.com/it/cmake/topic/862/iniziare-con-cmake

Capitolo 2: Aggiungi directory al percorso di inclusione del compilatore

Sintassi

- include_directories ([AFTER | BEFORE] [SYSTEM] dir1 [dir2 ...])

Parametri

Parametro	Descrizione
`dirN`	uno o più percorsi relativi o assoluti
`AFTER` , `BEFORE`	(facoltativo) se aggiungere le directory specificate all'inizio o alla fine dell'elenco corrente dei percorsi di inclusione; il comportamento predefinito è definito da `CMAKE_INCLUDE_DIRECTORIES_BEFORE`
`SYSTEM`	(facoltativo) indica al compilatore di utilizzare le directory specificate come *system include dirs* , che potrebbe attivare una gestione speciale da parte del compilatore

Examples

Aggiungi la sottodirectory di un progetto

Data la seguente struttura del progetto

```
include\
  myHeader.h
src\
  main.cpp
CMakeLists.txt
```

la seguente riga nel file `CMakeLists.txt`

```
include_directories(${PROJECT_SOURCE_DIR}/include)
```

aggiunge la directory `include` al *percorso* di *ricerca include* del compilatore per tutti i target definiti in questa directory (e tutte le relative sottodirectory incluse tramite `add_subdirectory()`).

Pertanto, il file `myHeader.h` nella sottodirectory `include` del progetto può essere incluso tramite `#include "myHeader.h"` nel file `main.cpp` .

Leggi Aggiungi directory al percorso di inclusione del compilatore online:

https://riptutorial.com/it/cmake/topic/5968/aggiungi-directory-al-percorso-di-inclusione-del-compilatore

Capitolo 3: Cerca e usa pacchetti, librerie e programmi installati

Sintassi

- find_package (pkgname [versione] [EXACT] [QUIET] [REQUIRED])
- includere (FindPkgConfig)
- pkg_search_module (prefisso [REQUIRED] [QUIET] pkgname [otherpkg ...])
- pkg_check_modules (prefisso [REQUIRED] [QUIET] pkgname [otherpkg ...])

Parametri

Parametro	Dettagli
versione (opzionale)	Versione minima del pacchetto definita da un numero maggiore e facoltativamente un numero minore, di patch e di tweak, nel formato major.minor.patch.tweak
ESATTO (facoltativo)	Specificare che la versione specificata nella `version` è la versione esatta da trovare
RICHIESTO (facoltativo)	Genera automaticamente un errore e interrompe il processo se il pacchetto non viene trovato
QUIET (opzionale)	La funzione non invierà alcun messaggio all'output standard

Osservazioni

- Il metodo `find_package` è compatibile su tutte le piattaforme, mentre la modalità `pkg-config` è disponibile solo su piattaforme tipo Unix, come Linux e OSX.
- Una descrizione completa del `find_package` numerosi parametri e opzioni nel manuale .
- Anche se è possibile specificare molti parametri facoltativi come la versione del pacchetto, non tutti i moduli di ricerca utilizzano correttamente tutti quei parametri. Se si verifica un comportamento non definito, potrebbe essere necessario trovare il modulo nel percorso di installazione di CMake e correggere o comprendere il suo comportamento.

Examples

Usa find_package e Trova Moduli .cmake

Il modo predefinito per trovare i pacchetti installati con CMake è utilizzare la funzione `find_package` insieme a un file `Find<package>.cmake` . Lo scopo del file è definire le regole di ricerca per il pacchetto e impostare variabili diverse, come `<package>_FOUND` , `<package>_INCLUDE_DIRS` e `<package>_LIBRARIES` .

Molti file `Find<package>.cmake` sono già definiti per impostazione predefinita in CMake. Tuttavia, se non è presente alcun file per il pacchetto necessario, puoi sempre scrivere il tuo e inserirlo in `${CMAKE_SOURCE_DIR}/cmake/modules` (o qualsiasi altra directory se `CMAKE_MODULE_PATH` stato sovrascritto)

Un elenco di moduli predefiniti è disponibile nel manuale (v3.6) . È essenziale controllare il manuale in base alla versione di CMake utilizzata nel progetto, altrimenti potrebbero mancare moduli. È anche possibile trovare i moduli installati con `cmake --help-module-list` .

C'è un bell'esempio per `FindSDL2.cmake` su Github

Ecco un `CMakeLists.txt` base che richiede SDL2:

```
cmake_minimum_required(2.8 FATAL_ERROR)
project("SDL2Test")

set(CMAKE_MODULE_PATH "${CMAKE_MODULE_PATH} ${CMAKE_SOURCE_DIR}/cmake/modules")
find_package(SDL2 REQUIRED)

include_directories(${SDL2_INCLUDE_DIRS})
add_executable(${PROJECT_NAME} main.c)
target_link_libraries(${PROJECT_NAME} ${SDL2_LIBRARIES})
```

Usa pkg_search_module e pkg_check_modules

Sui sistemi operativi Unix-like, è possibile utilizzare il programma `pkg-config` per trovare e configurare i pacchetti che forniscono un file `<package>.pc` .

Per poter utilizzare `pkg-config` , è necessario chiamare `include(FindPkgConfig)` in un `CMakeLists.txt` . Quindi, ci sono 2 possibili funzioni:

- `pkg_search_module` , che controlla il pacchetto e utilizza il primo disponibile.
- `pkg_check_modules` , che controlla tutti i pacchetti corrispondenti.

Ecco un `CMakeLists.txt` base che utilizza `pkg-config` per trovare SDL2 con versione precedente o uguale a 2.0.1:

```
cmake_minimum_required(2.8 FATAL_ERROR)
project("SDL2Test")

include(FindPkgConfig)
pkg_search_module(SDL2 REQUIRED sdl2>=2.0.1)

include_directories(${SDL2_INCLUDE_DIRS})
add_executable(${PROJECT_NAME} main.c)
target_link_libraries(${PROJECT_NAME} ${SDL2_LIBRARIES})
```

Leggi Cerca e usa pacchetti, librerie e programmi installati online:
https://riptutorial.com/it/cmake/topic/6752/cerca-e-usa-pacchetti--librerie-e-programmi-installati

Capitolo 4: Configura il file

introduzione

configure_file è una funzione di CMake per copiare un file in un'altra posizione e modificarne il contenuto. Questa funzione è molto utile per generare file di configurazione con percorsi, variabili personalizzate, utilizzando un modello generico.

Osservazioni

Copia un file in un'altra posizione e modifica il suo contenuto.

```
configure_file(<input> <output>
           [COPYONLY] [ESCAPE_QUOTES] [@ONLY]
           [NEWLINE_STYLE [UNIX|DOS|WIN32|LF|CRLF] ])
```

Copia un file in un file e sostituisce i valori delle variabili a cui si fa riferimento nel contenuto del file. Se è un percorso relativo, viene valutato rispetto alla directory sorgente corrente. Il file deve essere un file, non una directory. Se è un percorso relativo, viene valutato rispetto alla directory binaria corrente. Se si nomina una directory esistente, il file di input viene inserito in quella directory con il suo nome originale.

Se il file viene modificato, il sistema di generazione eseguirà nuovamente CMake per riconfigurare il file e generare nuovamente il sistema di generazione.

Questo comando sostituisce qualsiasi variabile nel file di input referenziato come $ {VAR} o @ VAR @ con i loro valori determinati da CMake. Se una variabile non è definita, verrà sostituita con nulla. Se viene specificato COPYONLY, non verrà eseguita alcuna espansione variabile. Se viene specificato ESCAPE_QUOTES, tutte le virgolette sostituite saranno in stile C con caratteri di escape. Il file verrà configurato con i valori correnti delle variabili CMake. Se viene specificato @ONLY, verranno sostituite solo le variabili del modulo @ VAR @ e $ {VAR} verrà ignorato. Questo è utile per la configurazione di script che usano $ {VAR}.

Le righe di file di input del modulo "#cmakedefine VAR ..." saranno sostituite con "#define VAR ..." o / * #undef VAR * / a seconda che VAR sia impostato su CMake su qualsiasi valore non considerato falso costante dal comando if (). (Il contenuto di "...", se presente, viene elaborato come sopra.) Le righe del file di input del modulo "# cmakedefine01 VAR" saranno sostituite con "#define VAR 1" o "#define VAR 0" in modo simile.

Con NEWLINE_STYLE è possibile regolare la fine della riga:

```
'UNIX' or 'LF' for \n, 'DOS', 'WIN32' or 'CRLF' for \r\n.
```

COPYONLY non deve essere utilizzato con NEWLINE_STYLE.

Examples

Genera un file di configurazione c ++ con CMake

Se abbiamo un progetto c ++ che usa un file di configurazione config.h con alcuni percorsi o variabili personalizzati, possiamo generarlo usando CMake e un file generico config.h.in.

Config.h.in può essere parte di un repository git, mentre il file generato config.h non verrà mai aggiunto, poiché viene generato dall'ambiente corrente.

```
#CMakeLists.txt
CMAKE_MINIMUM_REQUIRED(VERSION 2.8.11)

SET(PROJ_NAME "myproject")
PROJECT(${PROJ_NAME})

SET(${PROJ_NAME}_DATA     ""     CACHE PATH "This directory contains all DATA and RESOURCES")
SET(THIRDPARTIES_PATH    ${CMAKE_CURRENT_SOURCE_DIR}/../thirdparties      CACHE PATH "This
directory contains thirdparties")

configure_file ("${CMAKE_CURRENT_SOURCE_DIR}/common/config.h.in"
            "${CMAKE_CURRENT_SOURCE_DIR}/include/config.h" )
```

Se abbiamo un config.h.in in questo modo:

```
cmakedefine PATH_DATA "@myproject_DATA@"
cmakedefine THIRDPARTIES_PATH "@THIRDPARTIES_PATH@"
```

Le precedenti CMakeLists genereranno un header c ++ come questo:

```
#define PATH_DATA "/home/user/projects/myproject/data"
#define THIRDPARTIES_PATH "/home/user/projects/myproject/thirdparties"
```

Examble basato sulla versione di controllo SDL2

Se hai un modulo `cmake` . È possibile creare una cartella chiamata `in` per memorizzare tutti i file di configurazione.

Ad esempio, hai un progetto chiamato `FOO` , puoi creare un file `FOO_config.h.in` come:

```
//=============================================================================
//  CMake configuration file, based on SDL 2 version header
// ============================================================================

#pragma once

#include <string>
#include <sstream>

namespace yournamespace
{
  /**
```

```
 *  \brief Information the version of FOO_PROJECT in use.
 *
 *  Represents the library's version as three levels: major revision
 *  (increments with massive changes, additions, and enhancements),
 *  minor revision (increments with backwards-compatible changes to the
 *  major revision), and patchlevel (increments with fixes to the minor
 *  revision).
 *
 *  \sa FOO_VERSION
 *  \sa FOO_GetVersion
 */
typedef struct FOO_version
{
    int major;        /**< major version */
    int minor;        /**< minor version */
    int patch;        /**< update version */
} FOO_version;

/* Printable format: "%d.%d.%d", MAJOR, MINOR, PATCHLEVEL
*/
#define FOO_MAJOR_VERSION   0
#define FOO_MINOR_VERSION   1
#define FOO_PATCHLEVEL      0

/**
 *  \brief Macro to determine FOO version program was compiled against.
 *
 *  This macro fills in a FOO_version structure with the version of the
 *  library you compiled against. This is determined by what header the
 *  compiler uses. Note that if you dynamically linked the library, you might
 *  have a slightly newer or older version at runtime. That version can be
 *  determined with GUCpp_GetVersion(), which, unlike GUCpp_VERSION(),
 *  is not a macro.
 *
 *  \param x A pointer to a FOO_version struct to initialize.
 *
 *  \sa FOO_version
 *  \sa FOO_GetVersion
 */
#define FOO_VERSION(x)                          \
{                                       \
    (x)->major = FOO_MAJOR_VERSION;                 \
    (x)->minor = FOO_MINOR_VERSION;                 \
    (x)->patch = FOO_PATCHLEVEL;                    \
}

/**
 *  This macro turns the version numbers into a numeric value:
 *  \verbatim
    (1,2,3) -> (1203)
    \endverbatim
 *
 *  This assumes that there will never be more than 100 patchlevels.
 */
#define FOO_VERSIONNUM(X, Y, Z)                     \
    ((X)*1000 + (Y)*100 + (Z))

/**
 *  This is the version number macro for the current GUCpp version.
 */
#define FOO_COMPILEDVERSION \
```

```
    FOO_VERSIONNUM(FOO_MAJOR_VERSION, FOO_MINOR_VERSION, FOO_PATCHLEVEL)

/**
 *  This macro will evaluate to true if compiled with FOO at least X.Y.Z.
 */
#define FOO_VERSION_ATLEAST(X, Y, Z) \
    (FOO_COMPILEDVERSION >= FOO_VERSIONNUM(X, Y, Z))

}

// Paths
#cmakedefine FOO_PATH_MAIN "@FOO_PATH_MAIN@"
```

Questo file creerà un `FOO_config.h` nel percorso di installazione, con una variabile definita in c `FOO_PATH_MAIN` dalla variabile cmake. Per generarlo devi includere `in` file CMakeLists.txt, come questo (imposta percorsi e variabili):

```
MESSAGE("Configuring FOO_config.h ...")
configure_file("${CMAKE_CURRENT_SOURCE_DIR}/common/in/FOO_config.h.in"
"${FOO_PATH_INSTALL}/common/include/FOO_config.h" )
```

Tale file conterrà i dati del modello e variabile con il tuo percorso reale, ad esempio:

```
// Paths
#define FOO_PATH_MAIN "/home/YOUR_USER/Respositories/git/foo_project"
```

Leggi Configura il file online: https://riptutorial.com/it/cmake/topic/8304/configura-il-file

Capitolo 5: Costruisci configurazioni

introduzione

Questo argomento mostra gli usi di diverse configurazioni CMake come Debug o Release, in ambienti diversi.

Examples

Impostazione di una configurazione di rilascio / debug

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8.11)
SET(PROJ_NAME "myproject")
PROJECT(${PROJ_NAME})

# Configuration types
SET(CMAKE_CONFIGURATION_TYPES "Debug;Release" CACHE STRING "Configs" FORCE)
IF(DEFINED CMAKE_BUILD_TYPE AND CMAKE_VERSION VERSION_GREATER "2.8")
  SET_PROPERTY(CACHE CMAKE_BUILD_TYPE PROPERTY STRINGS  ${CMAKE_CONFIGURATION_TYPES})
ENDIF()

SET(${PROJ_NAME}_PATH_INSTALL     "/opt/project"                    CACHE PATH "This
directory contains installation Path")
SET(CMAKE_DEBUG_POSTFIX "d")

# Install
#-----------------------------------------------------#
INSTALL(TARGETS ${PROJ_NAME}
    DESTINATION  "${${PROJ_NAME}_PATH_INSTALL}/lib/${CMAKE_BUILD_TYPE}/"
    )
```

Eseguendo le seguenti build verranno generate due cartelle differenti ('/ opt / myproject / lib / Debug' / opt / myproject / lib / Release ') con le librerie:

```
$ cd /myproject/build
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
$ make
$ sudo make install
$ cmake _DCMAKE_BUILD_TYPE=Release ..
$ make
$ sudo make install
```

Leggi Costruisci configurazioni online: https://riptutorial.com/it/cmake/topic/8319/costruisci-configurazioni

Capitolo 6: Costruisci obiettivi

Sintassi

- add_executable (nome_destinazione [EXCLUDE_FROM_ALL] source1 [source2 ...])
- add_library (lib_name [STATIC | SHARED | MODULE] [EXCLUDE_FROM_ALL] source1 [source2 ...])

Examples

eseguibili

Per creare un target di build che produca un eseguibile, si dovrebbe usare il comando `add_executable` :

```
add_executable(my_exe
               main.cpp
               utilities.cpp)
```

Questo crea un target di build, ad esempio `make my_exe` per GNU make, con le invocazioni appropriate del compilatore configurato per produrre un file eseguibile `my_exe` dai due file di origine `main.cpp` e `utilities.cpp` .

Per impostazione predefinita, tutti i target eseguibili vengono aggiunti a `all` target incorporati (`all` per GNU make, `BUILD_ALL` per MSVC).
Per escludere un eseguibile da essere costruito con il target predefinito `all` , è possibile aggiungere il parametro opzionale `EXCLUDE_FROM_ALL` subito dopo il nome di destinazione:

```
add_executable(my_optional_exe EXCLUDE_FROM_ALL main.cpp)
```

biblioteche

Per creare un target di build che crea una libreria, utilizzare il comando `add_library` :

```
add_library(my_lib lib.cpp)
```

La variabile CMake `BUILD_SHARED_LIBS` controlla ogni volta che crea una libreria statica (`OFF`) o condivisa (`ON`), utilizzando ad esempio `cmake .. -DBUILD_SHARED_LIBS=ON` . Tuttavia, puoi impostare in modo esplicito la creazione di una libreria condivisa o statica aggiungendo `STATIC` o `SHARED` dopo il nome di destinazione:

```
add_library(my_shared_lib SHARED lib.cpp) # Builds an shared library
add_library(my_static_lib STATIC lib.cpp) # Builds an static library
```

Il file di output effettivo differisce tra i sistemi. Ad esempio, una libreria condivisa su sistemi Unix

viene solitamente chiamata `libmy_shared_library.so` , ma su Windows sarebbe `my_shared_library.dll` e `my_shared_library.lib` .

Come `add_executable` , aggiungi `EXCLUDE_FROM_ALL` prima dell'elenco dei file di origine per escluderlo da `all` target:

```
add_library(my_lib EXCLUDE_FROM_ALL lib.cpp)
```

Le librerie, progettate per essere caricate in fase di runtime (ad esempio plugin o applicazioni che utilizzano qualcosa come `dlopen`), dovrebbero usare `MODULE` invece di `SHARED` / `STATIC` :

```
add_library(my_module_lib MODULE lib.cpp)
```

Ad esempio, su Windows, non ci sarà un file di importazione (`.lib`), perché i simboli vengono esportati direttamente nel file `.dll` .

Leggi Costruisci obiettivi online: https://riptutorial.com/it/cmake/topic/3107/costruisci-obiettivi

Capitolo 7: Crea suite di test con CTest

Examples

Basic Test Suite

```
# the usual boilerplate setup
cmake_minimum_required(2.8)
project(my_test_project
        LANGUAGES CXX)

# tell CMake to use CTest extension
enable_testing()

# create an executable, which instantiates a runner from
# GoogleTest, Boost.Test, QtTest or whatever framework you use
add_executable(my_test
               test_main.cpp)

# depending on the framework, you need to link to it
target_link_libraries(my_test
                      gtest_main)

# now register the executable with CTest
add_test(NAME my_test COMMAND my_test)
```

La macro `enable_testing()` fa molta magia. Innanzitutto crea un `test` target incorporato (per GNU make, `RUN_TESTS` per VS), che, quando eseguito, esegue *CTest* .

La chiamata a `add_test()` registra finalmente un eseguibile arbitrario con *CTest* , quindi l'eseguibile viene eseguito ogni volta che chiamiamo il target di `test` .

Ora, crea il progetto come al solito e finalmente esegui il test target

GNU Make	Visual Studio
`make test`	`cmake --build . --target RUN_TESTS`

Leggi Crea suite di test con CTest online: https://riptutorial.com/it/cmake/topic/4197/crea-suite-di-test-con-ctest

Capitolo 8: Funzionalità di compilazione e selezione standard C / C ++

Sintassi

- target_compile_features (*target* **PRIVATE | PUBLIC | INTERFACE** *feature1* [*feature2* ...])

Examples

Compilare Requisiti di Caratteristica

Le caratteristiche del compilatore richieste possono essere specificate su una destinazione utilizzando il comando target_compile_features :

```
add_library(foo
    foo.cpp
)
target_compile_features(foo
    PRIVATE          # scope of the feature
    cxx_constexpr    # list of features
)
```

Le funzionalità devono essere parte di CMAKE_C_COMPILE_FEATURES o CMAKE_CXX_COMPILE_FEATURES ; altrimenti cmake segnala un errore. Cmake aggiungerà eventuali flag necessari come `-std=gnu++11` alle opzioni di compilazione del target.

Nell'esempio, le funzionalità sono dichiarate `PRIVATE` : i requisiti verranno aggiunti all'obiettivo, ma non ai suoi utenti. Per aggiungere automaticamente i requisiti a un edificio target contro foo, è necessario utilizzare `PUBLIC` o `INTERFACE` anziché `PRIVATE` :

```
target_compile_features(foo
    PUBLIC    # this time, required as public
    cxx_constexpr
)

add_executable(bar
    main.cpp
)
target_link_libraries(bar
    foo       # foo's public requirements and compile flags are added to bar
)
```

Selezione della versione C / C ++

La versione ricercata per C e C ++ può essere specificata globalmente utilizzando rispettivamente le variabili `CMAKE_C_STANDARD` (i valori accettati sono 98, 99 e 11) e `CMAKE_CXX_STANDARD` (i valori accettati sono 98, 11 e 14):

```
set(CMAKE_C_STANDARD 99)
set(CMAKE_CXX_STANDARD 11)
```

Questi aggiungeranno le opzioni di compilazione necessarie sugli obiettivi (es. `-std=c++11` per gcc).

La versione può essere resa obbligatoria impostando su `ON` le variabili `CMAKE_C_STANDARD_REQUIRED` e `CMAKE_CXX_STANDARD_REQUIRED` rispettivamente.

Le variabili devono essere impostate prima della creazione del target. La versione può anche essere specificata per target:

```
set_target_properties(foo PROPERTIES
    CXX_STANDARD 11
    CXX_STANDARD_REQUIRED ON
)
```

Leggi Funzionalità di compilazione e selezione standard C / C ++ online: https://riptutorial.com/it/cmake/topic/5297/funzionalita-di-compilazione-e-selezione-standard-c---c-plusplus

Capitolo 9: Funzioni e macro

Osservazioni

La principale differenza tra *macro* e *funzioni* è che le *macro* vengono valutate all'interno del contesto corrente, mentre le *funzioni* aprono un nuovo ambito all'interno di quello corrente. Pertanto, le variabili definite all'interno delle *funzioni* non sono note dopo che la funzione è stata valutata. Al contrario, le variabili all'interno dei *macro* sono ancora definite dopo che la macro è stata valutata.

Examples

Macro semplice per definire una variabile in base all'input

```
macro(set_my_variable _INPUT)
  if("${_INPUT}" STREQUAL "Foo")
    set(my_output_variable "foo")
  else()
    set(my_output_variable "bar")
  endif()
endmacro(set_my_variable)
```

Usa la macro:

```
set_my_variable("Foo")
message(STATUS ${my_output_variable})
```

stamperà

```
-- foo
```

mentre

```
set_my_variable("something else")
message(STATUS ${my_output_variable})
```

stamperà

```
-- bar
```

Macro per riempire una variabile di nome dato

```
macro(set_custom_variable _OUT_VAR)
  set(${_OUT_VAR} "Foo")
endmacro(set_custom_variable)
```

Usalo con

```
set_custom_variable(my_foo)
message(STATUS ${my_foo})
```

che stamperà

```
-- Foo
```

Leggi Funzioni e macro online: https://riptutorial.com/it/cmake/topic/2096/funzioni-e-macro

Capitolo 10: Integrazione di CMake negli strumenti di GitHub CI

Examples

Configura Travis CI con CMake di riserva

Travis CI ha preinstallato CMake 2.8.7.

Uno script `.travis.yml` minimo per una `.travis.yml` out-of source

```
language: cpp

compiler:
  - gcc

before_script:
  # create a build folder for the out-of-source build
  - mkdir build
  # switch to build directory
  - cd build
  # run cmake; here we assume that the project's
  # top-level CMakeLists.txt is located at '..'
  - cmake ..

script:
  # once CMake has done its job we just build using make as usual
  - make
  # if the project uses ctest we can run the tests like this
  - make test
```

Configura Travis CI con il nuovo CMake

La versione di CMake preinstallata su Travis è molto vecchia. Puoi usare i binari ufficiali di Linux per costruire con una versione più recente.

Ecco un esempio `.travis.yml` :

```
language: cpp

compiler:
  - gcc

# the install step will take care of deploying a newer cmake version
install:
  # first we create a directory for the CMake binaries
  - DEPS_DIR="${TRAVIS_BUILD_DIR}/deps"
  - mkdir ${DEPS_DIR} && cd ${DEPS_DIR}
  # we use wget to fetch the cmake binaries
  - travis_retry wget --no-check-certificate https://cmake.org/files/v3.3/cmake-3.3.2-Linux-
x86_64.tar.gz
```

```
  # this is optional, but useful:
  # do a quick md5 check to ensure that the archive we downloaded did not get compromised
  - echo "f3546812c11ce7f5d64dc132a566b749 *cmake-3.3.2-Linux-x86_64.tar.gz" > cmake_md5.txt
  - md5sum -c cmake_md5.txt
  # extract the binaries; the output here is quite lengthy,
  # so we swallow it to not clutter up the travis console
  - tar -xvf cmake-3.3.2-Linux-x86_64.tar.gz > /dev/null
  - mv cmake-3.3.2-Linux-x86_64 cmake-install
  # add both the top-level directory and the bin directory from the archive
  # to the system PATH. By adding it to the front of the path we hide the
  # preinstalled CMake with our own.
  - PATH=${DEPS_DIR}/cmake-install:${DEPS_DIR}/cmake-install/bin:$PATH
  # don't forget to switch back to the main build directory once you are done
  - cd ${TRAVIS_BUILD_DIR}

before_script:
  # create a build folder for the out-of-source build
  - mkdir build
  # switch to build directory
  - cd build
  # run cmake; here we assume that the project's
  # top-level CMakeLists.txt is located at '..'
  - cmake ..

script:
  # once CMake has done its job we just build using make as usual
  - make
  # if the project uses ctest we can run the tests like this
  - make test
```

Leggi Integrazione di CMake negli strumenti di GitHub CI online:
https://riptutorial.com/it/cmake/topic/1445/integrazione-di-cmake-negli-strumenti-di-github-ci

Capitolo 11: Packaging e progetti di distribuzione

Sintassi

- # Imballa una directory di compilazione
  pack [PATH]
- # Usa un generatore specifico
  cpack -G [GENERATORE] [PERCORSO]
- # Fornire sostituzioni opzionali
- cpack -G [GENERATORE] -C [CONFIGURAZIONE] -P [PACKAGE NAME] -R [PACKAGE VERSION] -B [PACKAGE DIRECTORY] --vendor [PACKAGE VENDOR]

Osservazioni

CPack è uno strumento esterno che consente il confezionamento rapido di progetti CMake creati raccogliendo tutti i dati richiesti direttamente dai file `CMakeLists.txt` e dai comandi di installazione utilizzati come `install_targets()` .

Perché CPack funzioni correttamente, `CMakeLists.txt` deve includere file o destinazioni da installare utilizzando la destinazione di `install` .

Uno script minimale potrebbe assomigliare a questo:

```
# Required headers
cmake(3.0)

# Basic project setup
project(my-tool)

# Define a buildable target
add_executable(tool main.cpp)

# Provide installation instructions
install_targets(tool DESTINATION bin)
```

Examples

Creazione di un pacchetto per un progetto CMake creato

Per creare un pacchetto ridistribuibile (ad esempio un archivio ZIP o un programma di installazione), di solito è sufficiente richiamare CPack usando una sintassi molto simile a quella di chiamare CMake:

```
cpack path/to/build/directory
```

A seconda dell'ambiente, questo raccoglierà tutti i file richiesti / installati per il progetto e li inserirà in un archivio compresso o in un programma di installazione autoestraente.

Selezione di un generatore CPack da utilizzare

Per creare un pacchetto utilizzando un formato specifico, è possibile scegliere il **generatore** da utilizzare.

Simile a CMake questo può essere fatto usando l'argomento **-G** :

```
cpack -G 7Z .
```

L'utilizzo di questa riga di comando consente di raggruppare il progetto creato nella directory corrente utilizzando il formato di archivio 7-Zip.

Al momento della scrittura, CPack versione 3.5 supporta i seguenti generatori per impostazione predefinita:

- Formato file `7Z` 7-Zip (archivio)
- `IFW` Qt Installer Framework (eseguibile)
- `NSIS` Null Soft Installer (eseguibile)
- `NSIS64` Null Soft Installer (64 bit, eseguibile)
- `STGZ` Tar GZip autoestraente (archivio)
- `TBZ2` Tar BZip2 (archivio)
- Compressione `TGZ` Tar GZip (archivio)
- `TXZ` Tar XZ (archivio)
- `TZ` Tar Compress compression (archivio)
- `WIX` formato file MSI tramite strumenti WiX (archivio eseguibile)
- Formato file `ZIP` ZIP (archivio)

Se non viene fornito alcun generatore esplicito, CPack cercherà di determinare il migliore disponibile a seconda dell'ambiente reale. Ad esempio, preferirà creare un eseguibile autoestraente su Windows e creare un archivio ZIP solo se non viene trovato un set di strumenti adeguato.

Leggi Packaging e progetti di distribuzione online:
https://riptutorial.com/it/cmake/topic/4368/packaging-e-progetti-di-distribuzione

Capitolo 12: Passi di costruzione personalizzati

introduzione

Le fasi di generazione personalizzate sono utili per eseguire target personalizzati nella creazione del progetto o per copiare facilmente i file in modo da non doverli eseguire manualmente (forse dll?). Qui vi mostrerò due esempi, il primo è per copiare dll (in particolare Qt5 dll) nella directory binaria dei vostri progetti (sia Debug che Release) e il secondo è per eseguire una destinazione personalizzata (Doxygen in questo caso) nella vostra soluzione (se stai usando Visual Studio).

Osservazioni

Come puoi vedere, puoi fare molto con obiettivi di build personalizzati e passaggi in cmake, ma dovresti stare attento ad usarli, specialmente quando copi le DLL. Sebbene sia conveniente farlo, a volte può risultare in quello che viene chiamato affettuosamente "inferno dll".

Fondamentalmente questo significa che puoi perderti in quale dll dipende il tuo eseguibile, quali sono i suoi caricamenti e quali devono essere eseguiti (forse a causa della variabile del tuo computer).

Oltre a questo avvertimento, sentiti libero di fare obiettivi personalizzati fare tutto ciò che vuoi! Sono potenti e flessibili e sono uno strumento inestimabile per qualsiasi progetto cmake.

Examples

Esempio di copia di Qt5 dll

Quindi diciamo che hai un progetto che dipende da Qt5 e devi copiare le dll rilevanti nella tua directory di costruzione e non vuoi farlo manualmente; puoi fare quanto segue:

```
cmake_minimum_required(VERSION 3.0)
project(MyQtProj LANGUAGES C CXX)
find_package(Qt5 COMPONENTS Core Gui Widgets)
#...set up your project

# add the executable
add_executable(MyQtProj ${PROJ_SOURCES} ${PROJ_HEADERS})

add_custom_command(TARGET MyQtProj POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Core>
$<TARGET_FILE_DIR:MyQtProj>
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Gui>
$<TARGET_FILE_DIR:MyQtProj>
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Widgets>
$<TARGET_FILE_DIR:MyQtProj>
    )
```

Così ora ogni volta che costruisci il tuo progetto, se le dll di destinazione sono cambiate che vuoi copiare, allora saranno copiate dopo che il tuo obiettivo (in questo caso l'eseguibile principale) è stato costruito (nota il comando `copy_if_different`); in caso contrario, non verranno copiati.

Inoltre, si noti l'uso delle espressioni del generatore qui. Il vantaggio nell'usarli è che non devi dire esplicitamente dove copiare le DLL o quali varianti usare. Per poterli usare, però, il progetto che stai usando (Qt5 in questo caso) deve avere obiettivi importati.

Se si sta eseguendo il debug, CMake sa (in base alla destinazione importata) copiare i file Qt5Cored.dll, Qt5Guid.dll e Qt5Widgetsd.dll nella cartella Debug della cartella di generazione. Se stai costruendo in versione, le versioni di rilascio di .dlls verranno copiate nella cartella di rilascio.

Esecuzione di una destinazione personalizzata

È anche possibile creare un target personalizzato da eseguire quando si desidera eseguire una determinata attività. Questi sono in genere eseguibili eseguiti per fare cose diverse. Qualcosa che può essere di particolare utilità è eseguire Doxygen per generare documentazione per il tuo progetto. Per fare questo puoi fare quanto segue nel tuo `CMakeLists.txt` (per semplicità continueremo il nostro esempio di progetto Qt5):

```
cmake_minimum_required(VERSION 3.0)
project(MyQtProj LANGUAGES C CXX)
find_package(Qt5 COMPONENTS Core Gui Widgets)
#...set up your project

add_executable(MyQtProj ${PROJ_SOURCES} ${PROJ_HEADERS})

add_custom_command(TARGET MyQtProj POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Core>
$<TARGET_FILE_DIR:MyQtProj>
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Gui>
$<TARGET_FILE_DIR:MyQtProj>
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Widgets>
$<TARGET_FILE_DIR:MyQtProj>
    )

#Add target to build documents from visual studio.
set(DOXYGEN_INPUT ${CMAKE_CURRENT_SOURCE_DIR}/Doxyfile)
#set the output directory of the documentation
set(DOXYGEN_OUTPUT_DIR ${CMAKE_CURRENT_SOURCE_DIR}/docs)
# sanity check...
message("Doxygen Output ${DOXYGEN_OUTPUT_DIR}")
find_package(Doxygen)

if(DOXYGEN_FOUND)
    # create the output directory where the documentation will live
    file(MAKE_DIRECTORY ${DOXYGEN_OUTPUT_DIR})
    # configure our Doxygen configuration file. This will be the input to the doxygen
    # executable
    configure_file(${CMAKE_CURRENT_SOURCE_DIR}/Doxyfile.in
    ${CMAKE_CURRENT_BINARY_DIR}/Doxyfile @ONLY)

# now add the custom target. This will create a build target called 'DOCUMENTATION'
# in your project
ADD_CUSTOM_TARGET(DOCUMENTATION
```

```
  COMMAND ${CMAKE_COMMAND} -E echo_append "Building API Documentation..."
  COMMAND ${CMAKE_COMMAND} -E make_directory ${DOXYGEN_OUTPUT_DIR}
  COMMAND ${DOXYGEN_EXECUTABLE} ${CMAKE_CURRENT_BINARY_DIR}/Doxyfile
    WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
  COMMAND ${CMAKE_COMMAND} -E echo "Done."
  WORKING_DIRECTORY ${DOXYGEN_OUTPUT_DIR})

endif(DOXYGEN_FOUND)
```

Ora quando creiamo la nostra soluzione (sempre supponendo che tu stia usando Visual Studio), avrai un obiettivo di compilazione chiamato `DOCUMENTATION` che puoi costruire per rigenerare la documentazione del tuo progetto.

Leggi Passi di costruzione personalizzati online: https://riptutorial.com/it/cmake/topic/9537/passi-di-costruzione-personalizzati

Capitolo 13: Progetto gerarchico

Examples

Approccio semplice senza pacchetti

Esempio che costruisce un eseguibile (editor) e collega una libreria (evidenziata) ad esso. La struttura del progetto è semplice, ha bisogno di un master CMakeLists e di una directory per ogni sottoprogetto:

```
CMakeLists.txt
editor/
    CMakeLists.txt
    src/
        editor.cpp
highlight/
    CMakeLists.txt
    include/
        highlight.h
    src/
        highlight.cpp
```

Il master CMakeLists.txt contiene le definizioni globali e la chiamata `add_subdirectory` per ciascun sottoprogetto:

```
cmake_minimum_required(VERSION 3.0)
project(Example)

add_subdirectory(highlight)
add_subdirectory(editor)
```

CMakeLists.txt per la libreria assegna le fonti e include le directory. Usando `target_include_directories()` invece di `include_directories()` le directory di inclusione verranno propagate agli utenti della libreria:

```
cmake_minimum_required(VERSION 3.0)
project(highlight)

add_library(${PROJECT_NAME} src/highlight.cpp)
target_include_directories(${PROJECT_NAME} PUBLIC include)
```

CMakeLists.txt per l'applicazione assegna le fonti e collega la libreria di evidenziazione. I percorsi per il binario di hightlighter e include sono gestiti automaticamente da cmake:

```
cmake_minimum_required(VERSION 3.0)
project(editor)

add_executable(${PROJECT_NAME} src/editor.cpp)
target_link_libraries(${PROJECT_NAME} PUBLIC highlight)
```

Leggi Progetto gerarchico online: https://riptutorial.com/it/cmake/topic/1443/progetto-gerarchico

Capitolo 14: Test e debug

Examples

Approccio generale per eseguire il debug quando si crea con Make

Supponiamo che la `make` fallisca:

```
$ make
```

Lanciatelo invece con `make VERBOSE=1` per vedere i comandi eseguiti. Quindi esegui direttamente il comando linker o il compilatore che vedrai. Cerca di farlo funzionare aggiungendo le bandiere o le librerie necessarie.

Quindi scopri cosa cambiare, quindi CMake stesso può passare argomenti corretti al comando compiler / linker:

- cosa cambiare nel sistema (quali librerie installare, quali versioni, versioni di CMake stessa)
- se precedente fallisce, quali variabili d'ambiente impostare o parametri da passare a CMake
- in caso contrario, cosa cambiare nel file `CMakeLists.txt` o negli script di rilevamento della libreria come `FindSomeLib.cmake`

Per aiutare in questo, aggiungi le chiamate del `message(${MY_VARIABLE})` in `CMakeLists.txt` o `*.cmake` per eseguire il debug delle variabili che vuoi controllare.

Lascia che CMake crei Makefile dettagliati

Una volta che un progetto CMake viene inizializzato tramite `project()` , la verbosità di output dello script di build risultante può essere regolata tramite:

```
CMAKE_VERBOSE_MAKEFILE
```

Questa variabile può essere impostata tramite la riga di comando di CMake quando si configura un progetto:

```
cmake -DCMAKE_VERBOSE_MAKEFILE=ON <PATH_TO_PROJECT_ROOT>
```

Per GNU rendere questa variabile ha lo stesso effetto dell'esecuzione `make VERBOSE=1` .

Debug find_package () errori

Nota: i messaggi di errore di CMake mostrati includono già la correzione per percorsi di installazione di librerie / strumenti "non standard". Gli esempi seguenti dimostrano solo `find_package()` CMake `find_package()` più dettagliati.

CMake Package / Module supportato internamente

Se il seguente codice (sostituire il modulo `FindBoost` con il modulo in questione)

```
cmake_minimum_required(VERSION 2.8)
project(FindPackageTest)

find_package(Boost REQUIRED)
```

dà qualche errore come

```
CMake Error at [...]/Modules/FindBoost.cmake:1753 (message):
  Unable to find the requested Boost libraries.

  Unable to find the Boost header files.  Please set BOOST_ROOT to the root
  directory containing Boost or BOOST_INCLUDEDIR to the directory containing
  Boost's headers.
```

E ti stai chiedendo dove ha provato a trovare la libreria, puoi controllare se il tuo pacchetto ha un'opzione `_DEBUG` come il modulo `Boost` ha per ottenere un output più dettagliato

```
$ cmake -D Boost_DEBUG=ON ..
```

CMake ha abilitato il pacchetto / libreria

Se il seguente codice (sostituire il `Xyz` con la tua biblioteca in questione)

```
cmake_minimum_required(VERSION 2.8)
project(FindPackageTest)

find_package(Xyz REQUIRED)
```

dà qualche errore come

```
CMake Error at CMakeLists.txt:4 (find_package):
  By not providing "FindXyz.cmake" in CMAKE_MODULE_PATH this project has
  asked CMake to find a package configuration file provided by "Xyz", but
  CMake did not find one.

  Could not find a package configuration file provided by "Xyz" with any of
  the following names:

    XyzConfig.cmake
    xyz-config.cmake

  Add the installation prefix of "Xyz" to CMAKE_PREFIX_PATH or set "Xyz_DIR"
  to a directory containing one of the above files.  If "Xyz" provides a
  separate development package or SDK, be sure it has been installed.
```

E ti stai chiedendo dove ha provato a trovare la libreria, puoi usare la variabile globale non documentata `CMAKE_FIND_DEBUG_MODE` per ottenere un output più dettagliato

```
$ cmake -D CMAKE_FIND_DEBUG_MODE=ON ..
```

Leggi Test e debug online: https://riptutorial.com/it/cmake/topic/4098/test-e-debug

Capitolo 15: Utilizzo di CMake per configurare i tag preproprocessore

introduzione

L'uso di CMake in un progetto C ++ se usato correttamente può consentire al programmatore di concentrarsi meno sulla piattaforma, sul numero di versione del programma e altro sul programma stesso. Con CMake è possibile definire i tag del preprocessore che consentono di verificare facilmente la piattaforma o qualsiasi altro tag del preprocessore che potrebbe essere necessario nel programma effettivo. Come il numero di versione che potrebbe essere sfruttato in un sistema di log.

Sintassi

- #define preprocessor_name "@ cmake_value @"

Osservazioni

È importante capire che non tutti i preprocessori dovrebbero essere definiti in `config.h.in` . I tag del preprocessore sono generalmente utilizzati solo per semplificare la vita dei programmatori e dovrebbero essere utilizzati con discrezione. Dovresti cercare se esiste già un tag per il preprocessore prima di definirlo in quanto potresti imbatterti in comportamenti non definiti su sistemi diversi.

Examples

Utilizzo di CMake per definire il numero di versione per l'utilizzo di C ++

Le possibilità sono infinite. come puoi usare questo concetto per estrarre il numero di versione dal tuo sistema di compilazione; come git e usa quel numero di versione nel tuo progetto.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.8)
project(project_name VERSION "0.0.0")

configure_file(${path to configure file 'config.h.in'}
include_directories(${PROJECT_BINARY_BIN}) // this allows the 'config.h' file to be used
throughout the program

...
```

config.h.in

```
#ifndef INCLUDE_GUARD
#define INCLUDE_GUARD

#define PROJECT_NAME "@PROJECT_NAME@"
#define PROJECT_VER  "@PROJECT_VERSION@"
#define PROJECT_VER_MAJOR "@PROJECT_VERSION_MAJOR@"
#define PROJECT_VER_MINOR "@PROJECT_VERSION_MINOR@"
#define PTOJECT_VER_PATCH "@PROJECT_VERSION_PATCH@"

#endif // INCLUDE_GUARD
```

main.cpp

```
#include <iostream>
#include "config.h"
int main()
{
    std::cout << "project name: " << PROJECT_NAME << " version: " << PROJECT_VER << std::endl;
    return 0;
}
```

produzione

```
project name: project_name version: 0.0.0
```

Leggi Utilizzo di CMake per configurare i tag preproprocessore online: https://riptutorial.com/it/cmake/topic/10885/utilizzo-di-cmake-per-configurare-i-tag-preproprocessore

Capitolo 16: Variabili e proprietà

introduzione

La semplicità delle variabili CMake di base smentisce la complessità della sintassi completa delle variabili. Questa pagina documenta i vari casi variabili, con esempi, e sottolinea le insidie da evitare.

Sintassi

- set (nome_name variabile [descrizione tipo CACHE [FORCE]]))

Osservazioni

I nomi delle variabili fanno distinzione tra maiuscole e minuscole. I loro valori sono di tipo stringa. Il valore di una variabile è referenziato tramite:

```
${variable_name}
```

e viene valutato all'interno di un argomento citato

```
"${variable_name}/directory"
```

Examples

Variabile cache (globale)

```
set(my_global_string "a string value"
    CACHE STRING "a description about the string variable")
set(my_global_bool TRUE
    CACHE BOOL "a description on the boolean cache entry")
```

Nel caso in cui una variabile memorizzata nella cache sia già definita nella cache quando CMake elabora la rispettiva riga (ad esempio quando viene eseguito nuovamente Esegui CMake), non viene alterata. Per sovrascrivere il valore predefinito, aggiungi `FORCE` come ultimo argomento:

```
set(my_global_overwritten_string "foo"
    CACHE STRING "this is overwritten each time CMake is run" FORCE)
```

Variabile locale

```
set(my_variable "the value is a string")
```

Per impostazione predefinita, una variabile locale viene definita solo nella directory corrente e in qualsiasi sottodirectory aggiunta tramite il comando `add_subdirectory` .

Per estendere l'ambito di una variabile ci sono due possibilità:

1. `CACHE` , che lo renderà globalmente disponibile
2. usa `PARENT_SCOPE` , che lo renderà disponibile nell'ambito genitore. L'ambito genitore è il file `CMakeLists.txt` nella directory padre o il chiamante della funzione corrente.

   Tecnicamente la directory principale sarà il file `CMakeLists.txt` che includeva il file corrente tramite il comando `add_subdirectory` .

Archi ed elenchi

È importante sapere in che modo CMake distingue tra elenchi e stringhe semplici. Quando scrivi:

```
set(VAR "ab c")
```

crei una **stringa** con il valore `"ab c"` . Ma quando scrivi questa frase senza virgolette:

```
set(VAR abc)
```

Si crea invece un **elenco** di tre elementi: `"a"` , `"b"` e `"c"` .

Le variabili non-list sono in realtà anche liste (di un singolo elemento).

Le liste possono essere gestite con il comando `list()` , che consente di concatenare liste, cercarle, accedere a elementi arbitrari e così via (documentazione di list ()).

Un po 'di confusione, una **lista** è anche una **stringa** . La linea

```
set(VAR abc)
```

è equivalente a

```
set(VAR "a;b;c")
```

Pertanto, per concatenare gli elenchi si può anche usare il comando `set()` :

```
set(NEW_LIST "${OLD_LIST1};${OLD_LIST2})"
```

Variabili e cache delle variabili globali

Principalmente userete "variabili normali" :

```
set(VAR TRUE)
set(VAR "main.cpp")
set(VAR1 ${VAR2})
```

Ma CMake conosce anche le "variabili memorizzate nella cache" globali (persistono in `CMakeCache.txt`). E se nello scope corrente esistono variabili normali e memorizzate nella cache

con lo stesso nome, le variabili normali nascondono quelle memorizzate nella cache:

```
cmake_minimum_required(VERSION 2.4)
project(VariablesTest)

set(VAR "CACHED-init" CACHE STRING "A test")
message("VAR = ${VAR}")

set(VAR "NORMAL")
message("VAR = ${VAR}")

set(VAR "CACHED" CACHE STRING "A test" FORCE)
message("VAR = ${VAR}")
```

Uscita della prima uscita

```
VAR = CACHED-init
VAR = NORMAL
VAR = CACHED
```

Uscita della seconda uscita

```
VAR = CACHED
VAR = NORMAL
VAR = CACHED
```

Nota: l'opzione `FORCE` disattiva / rimuove la variabile normale dall'ambito corrente.

Usa casi per le variabili memorizzate nella cache

Di solito ci sono due casi d'uso (si prega di non utilizzarli impropriamente per variabili globali):

1. Un valore nel tuo codice dovrebbe essere modificabile dall'utente del tuo progetto, ad es. Con l' `cmakegui` , `ccmake` o con `cmake -D ...` :

   CMakeLists.txt / MyToolchain.cmake

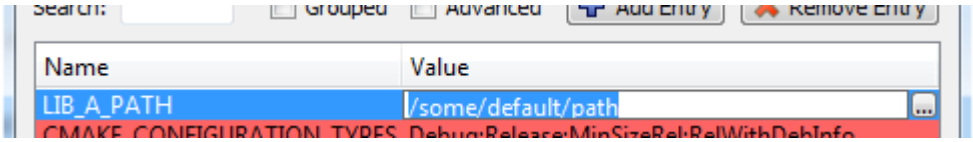
   ```
   set(LIB_A_PATH "/some/default/path" CACHE PATH "Path to lib A")
   ```

   Riga di comando

   ```
   $ cmake -D LIB_A_PATH:PATH="/some/other/path" ..
   ```

   Questo pre-imposta questo valore nella cache e la riga sopra non la modificherà.

   CMake GUI



Nella GUI l'utente avvia per prima cosa il processo di configurazione, quindi può modificare qualsiasi valore memorizzato nella cache e termina con l'avvio della generazione dell'ambiente di generazione.

2. Inoltre, CMake esegue la cache dei risultati di identificazione ricerca / test / compilatore (quindi non è necessario ripetere l'operazione ogni volta che si rieseguono i passaggi di configurazione / generazione)

```
find_path(LIB_A_PATH libA.a PATHS "/some/default/path")
```

Qui `LIB_A_PATH` viene creato come variabile memorizzata nella cache.

Aggiunta di flag di profilazione a CMake per utilizzare gprof

La serie di eventi qui dovrebbe funzionare come segue:

1. Compila il codice con l'opzione -pg
2. Codice di collegamento con opzione -pg
3. Eseguire il programma
4. Il programma genera il file gmon.out
5. Esegui il programma gprof

Per aggiungere flag di profilatura, devi aggiungere a CMakeLists.txt:

```
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pg")
SET(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -pg")
SET(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -pg")
```

Questo deve aggiungere flag per compilare e collegare e utilizzare dopo aver eseguito il programma:

gprof ./my_exe

Se ricevi un errore del tipo:

```
gmon.out: No such file or directory
```

Ciò significa che la compilazione non ha aggiunto correttamente le informazioni di profilazione.

Leggi Variabili e proprietà online: https://riptutorial.com/it/cmake/topic/2091/variabili-e-proprieta

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con cmake	Amani, arrowd, ComicSansMS, Community, Daniel Schepler, dontloo, Fantastic Mr Fox, fedepad, Florian, greatwolf, Mario, Neui, OliPro007, Torbjörn, Ziv
2	Aggiungi directory al percorso di inclusione del compilatore	kiki, Torbjörn
3	Cerca e usa pacchetti, librerie e programmi installati	OliPro007
4	Configura il file	Jav_Rock, Shihe Zhang, vgonisanz
5	Costruisci configurazioni	Jav_Rock
6	Costruisci obiettivi	arrowd, Neui, Torbjörn
7	Crea suite di test con CTest	arrowd, ComicSansMS, Torbjörn
8	Funzionalità di compilazione e selezione standard C / C ++	wasthishelpful
9	Funzioni e macro	Torbjörn
10	Integrazione di CMake negli strumenti di GitHub CI	ComicSansMS
11	Packaging e progetti di distribuzione	Mario, Meysam, Neui
12	Passi di costruzione personalizzati	Developer Paul
13	Progetto gerarchico	Adam Trhon, Anedar, Clare Macrae, Robert

14	Test e debug	Florian, Torbjörn, Velkan
15	Utilizzo di CMake per configurare i tag preproprocessore	JVApen, Matthew
16	Variabili e proprietà	arrowd, CivFan, Florian, Torbjörn, Trilarion, Trygve Laugstøl, vgonisanz